DOCUMENT RESUME

ED 135 377 IR 004 486

AUTHOR	Francis, Larry
TITLE	The Tutor Training Course: Lessons Learned.
INSTITUTION	Illinois Univ., Urbana. Computer-Based Education Lab.
SPONS AGENCY	Advanced Research Projects Agency (DOD), Washington, D.C.; National Science Foundation, Washington, D.C.
PUB DATE	76
CONTRACT	DAHC-15-73-C-0077; US-NSF-C-723
NOTE	139p.
EDRS PRICE	MF-$0.83 HC-$7.35 Plus Postage.
DESCRIPTORS	*Computer Assisted Instruction; Computer Programs; Computers; *Instructional Design; Instructional Materials; Instructional Systems; Instructional Technology; On Line Systems; Programers; *Programing; *Programing Languages; Training Objectives
IDENTIFIERS	PLATO IV; Programmed Logic for Automatic Teaching Operations; *TUTOR

ABSTRACT

The first formal author training course for the Tutor programing language and the use of the PLATO system was designed and conducted by the Military Training Centers (MTC) group. The course was developed according to thirteen cognitive and affective principles, and was used over a period of three years to train approximately 100 authors. This report contains a statement of the principles and a description of their implementation, including many examples from course materials. It also recounts the highlights and turning points of the author training course, reviews the basis for its modification, and examines the dilemmas encountered in teaching new authors to prepare computer-based instruction materials. Techniques for resolving some of these dilemmas are suggested. Also included is the course feedback from outside groups. This report is directed to instructors of new authors, developers of author training materials, and managers of computer-based instruction development centers. (Author/SC)

ED135377

Computer-based Education

Research Laboratory

University of Illinois — Urbana Illinois

Bill Strutz

THE TUTOR TRAINING COURSE:

LESSONS LEARNED

LARRY FRANCIS

IR004486

THE TUTOR TRAINING COURSE: LESSONS LEARNED

Larry Francis

Illustrated by

Wayne Wilson

COMPUTER-BASED EDUCATION RESEARCH LABORATORY
UNIVERSITY OF ILLINOIS, Urbana-Champaign

This research was supported in part by Advanced Research Projects Agency of the Department of Defense under U.S. Army Contract DAHC-15-73-C-0077 and indirectly by the National Science Foundation (US NSF C-723).

". . . we wish to apologize to our women readers for our use, in this book, of a sexist grammatical convention. We were unable to find or invent a stylistically graceful substitute for the pronouns he and him in instances where we obviously mean to refer to both male and female."

From Neil Postman and Charles Weingartner, The School Book (New York, N.Y.: Dell Publishing Co., 1973), p. v.

Acknowledgement

Many people have contributed critiques, programming, advice, and even whole lessons to the MTC training package. In addition to a great deal of thanks to members of the current MTC and PEER group staffs (Alec Himwich, Joe Klecka, Lynn Misselt, Eileen Sweeney, Allen Avner, and John Gilpin), I would like to thank many others for their role in developing the training course and preparing this report. In the order of their contribution, they are: Robert Bohn, David Meller, Daniel Hyde, Craig Burson, Pauline Jordan, Mary Jane Hyde, James Kraatz, and Elaine Avner.

I would also like to thank Kathy Geissler and Julie Garrard for typing the manuscript and Wayne Wilson for providing illustrations.

Abstract

The Military Training Centers (MTC) Group created and taught the first formal author training course for the TUTOR programming language and the use of the PLATO system. The course was used over a period of three years to train approximately 100 authors who spent two to three weeks at the Computer-based Education Research Laboratory (CERL) of the University of Illinois at Urbana-Champaign. In general, the new authors had little previous experience with computers or programmed instruction; however, many were classroom instructors.

Thirteen cognitive and affective principles guided the creation of the author course; five were posited originally, and the rest were formulated based on experience derived from teaching the course. This report contains a statement of the principles and a description of their implementation, including many examples from course materials. It also recounts the highlights and turning points of the author training course, reviews the basis for its modification, and examines the dilemmas encountered in teaching new authors to prepare computer-based instruction materials. Techniques for resolving some of these dilemmas are suggested.

The MTC training course was evaluated by examining the experience of users outside of MTC. The opinions and recommendations of these outside groups suggest that they found the materials valuable and effective. This report is directed to instructors of new authors, developers of author training materials, and managers of computer-based instruction development centers.

Table of Contents

Introduction

During the past four years, the Military Training Centers (MTC) Group has created and used a set of training materials for teaching basic TUTOR to new authors at ARPA/PLATO sites. The purpose of the training was to provide each new author with enough basic skills so that whatever else was needed could be learned by self-study and experimentation.

About 125 ARPA authors have learned TUTOR from the materials; the training of about 2/3 of these authors was directly supervised by MTC staff. Many non-ARPA users also found the materials useful and obtained copies. We estimate that at least 125 authors were trained by other instructors who used our materials.

This report describes the lessons learned in that effort. The reader of this report is assumed to have familiarity with the PLATO system, the TUTOR language, and the role, purpose, and function of the MTC group. The latter is best described in the series of annual and semi-annual reports which cover all of the ARPA/PLATO activities.[1] The reader is also assumed to be familiar with the terminology and concepts of Computer-Based Education (CBE).

---

[1] See for example "Demonstration and Evaluation of the PLATO IV Computer-based Education System," First Annual Report for the period August 2, 1972-January 1, 1974, Computer-based Education Research Laboratory, University of Illinois, Urbana-Champaign, February, 1974.

MTC also developed as part of the new author training course a "mini-course" on instructional design for CBE. The development of this course will be described in a later report.

This report is directed to the interests of instructors of new authors, developers of author training materials, and managers of computer-based instruction development centers. Knowledge of the TUTOR language is helpful, but not necessary, for understanding this report. In cases when concepts have been clarified with specific examples, readers unfamiliar with TUTOR can skip these comments with little loss.

The Principles of MTC TUTOR Training

Overview

The principles by which the MTC group has designed its training are based on experience gained while teaching the PLATO basic authoring course, providing follow-up training, and consulting with the ARPA/PLATO authors during the courseware development phase of their projects.

Most of the authors MTC trained had never before worked with a computer and had little or no conception of Computer-Assisted Instruction (CAI), Computer-Based Education (CBE), or Computer-Managed Instruction (CMI). They had not seen a PLATO author at work, viewed an operating CBE classroom, nor learned from a CBE system. Few had experience with other "packaged instruction" media, e.g., programmed instruction, video casettes, film strips. Moreover, the hardware used for the PLATO system was new even to those users familiar with CBE. Thus, the MTC author training course had to provide basic instruction in computer science, CBE, and programmed learning, as well as instruction in writing PLATO lessons. Although such subjects as instructional design, CBE site management, hardware trouble shooting, and peripheral equipment utilization were included in the training program, this report will focus only on TUTOR training.

TUTOR training is clearly one of the most important and most visible components of the MTC training package. In fact, upon arriving at the Computer-based Education Research Laboratory (CERL), new ARPA/PLATO authors consistently viewed learning TUTOR as the only objective of the course.

Objectives

The MTC TUTOR training course attempted to accomplish three types of objectives:

1. Cognitive. Facts, conventions, system characteristics were taught (e.g. TUTOR command formats, hardware capacities and limits). In general, these items require little interpretation, reflection, or analysis.

2. Cognitive/Affective. Authoring tips and useful procedures (i.e., items which are based on rules derived by use rather than on facts or theory) were presented. Typically, these procedures are "better" or "best" ways an experienced group of individuals has found to code an algorithm, find information, or organize a task.

3. Affective. Changed or new attitudes towards computers, CBE, and authoring styles (i.e., developing a model for typical author behavior) were encouraged. The information on which these objectives are based is subjective, and moreover, relates not to the specific details of using the PLATO system, but rather to the delineation of the role and attitude of an author. We need to emphasize that few, if any, of the authors on the PLATO system have ever viewed it from the student's perspective: they have not been required to earn credits or grades based on their understanding of a PLATO lesson.

By strict definition, all of the items suggested by "3" above and some of those listed for "2" would not be included in a description of TUTOR training. However, as we have already noted, training the MTC group provided was a combination of TUTOR training, instructional design training, basic computer science, and acclimatization to CBE. Any description of our TUTOR training course requires the drawing of arbitrary lines to define TUTOR training as a subset of overall author training. Because MTC interlaced various types of training, we often accomplished, for example, instructional design objectives while overtly teaching TUTOR. For that reason, the objectives listed above and the principles given

below are not only those directly related to the teaching of TUTOR, but also include those followed during the portion of the course directly concerned with TUTOR training. At the end of the list of principles there is an index to some examples of the ways that the principles were implemented in the MTC training course.

Thirteen Principles

The principles which the MTC basic author training course attempted to adhere to are as follows:

1. Introduce information in small segments (typically one to three new commands, or the equivalent, at a time).

   a. Provide exercises for each aspect of each command before introducing new aspects, i.e., distribute practice.

   b. Provide correct "solutions" for the exercises to remove misconceptions the student may form about language usage before they adversely affect the new author's work.

2. Test the student often (after each segment).

   a. Test command usage, function, execution, and placement (i.e., the order of commands) rather than command formats or syntax.

   b. When possible, test via problems that permit multiple solutions--to avoid overly-contrived questions.

   c. Test via computer-generated problems so that mastery learning and generalization may be achieved.

3. Demonstrate a need for a feature, then introduce the solution (a new command, key, etc.). This method helps to insure that the author is motivated to learn and use his new knowledge immediately. To amplify, do not teach "all about" a new topic when it is first introduced. That will only provide the author with solutions to problems he has not learned to recognize.

4. Intermix teaching about different facets of the PLATO system (commands, editing features, author behavior, keys), rather than teaching all about one aspect at once.

5. Self-pace the course.

6. Teach each author different TUTOR specialized subtopics (e.g., managing a student roster, setting up a router, graphing, character sets). This technique solves five problems:

   Training time is typically too short to teach each author all topics.

   Authors need enough time to practice what they have learned. Without practice, the information overload resulting from learning too much, too fast will be counterproductive. Teaching each author a few subtopics gives him time to practice them immediately. Later he can learn and practice other subtopics by communicating with other authors who took the training at the same time.

   In a self-paced course, the unequal abilities among authors can cause frustration for slower authors, but insertion of subtopic training, matched in complexity and duration to each author's progress, can keep the group "bunched."

   Skills in the subtopic areas are needed nearly immediately on-site.

   Authors should begin to learn to assist and depend upon each other. In fact, each author can be an on-site consultant for the rest of the group and can feel comfortable with at least one "advanced" technique.

7. Reduce the proctoring needed by providing an on-line reference manual. Rather than putting all the information in the form of long readings, we provided a brief overview which was followed by a "look-up" in the on-line manual. Thus the authors interact with on-line materials in much the same way that students interact with a lesson. This approach also forms in the author the habit of using the system as an information source.

8. For authors unacquainted with computers, introduce unfamiliar ideas (e.g., looping) via many small steps leading ultimately to a thorough understanding of the principles of operation.

9. Encourage experimentation to discover results of undocumented systems features. Some authors need to be reassured that their experiments or mistakes are not harmful to the system and that a "try it and see" approach is often the best one. In this way it is often possible to convert potential anxiety into curiosity--a more positive attitude. For this same reason, we left some details about command operations unspecified--for the new author to discover.

10. Teach via a PLATO lesson several important facts and concepts which are available nowhere else. In this way, the author's curiosity and motivation to learn "how they did that" is enhanced. Also, the author will become aware of certain esthetic and instructional design problems since he will feel like a real student,

with the attendant problems and frustrations. (Realized mid-way into the project, this principle was not incorporated everywhere it could and should have been.)

11. Present models of successful authors: how they proceed; how frequently they interact with other authors, the terminal, and on-line colleagues; their need for documentation; and their attitudes about innovation, plagiarism, and game playing.

12. Follow the basic training with retraining and advanced topics training on-site at a later date.

13. Anticipate problems which the new author is unlikely to encounter until after formal training and provide him with exercises that will allow him to avoid or solve these problems. For example, a new author may not be easily convinced of the need for good documentation by mere exhortation from his instructor.

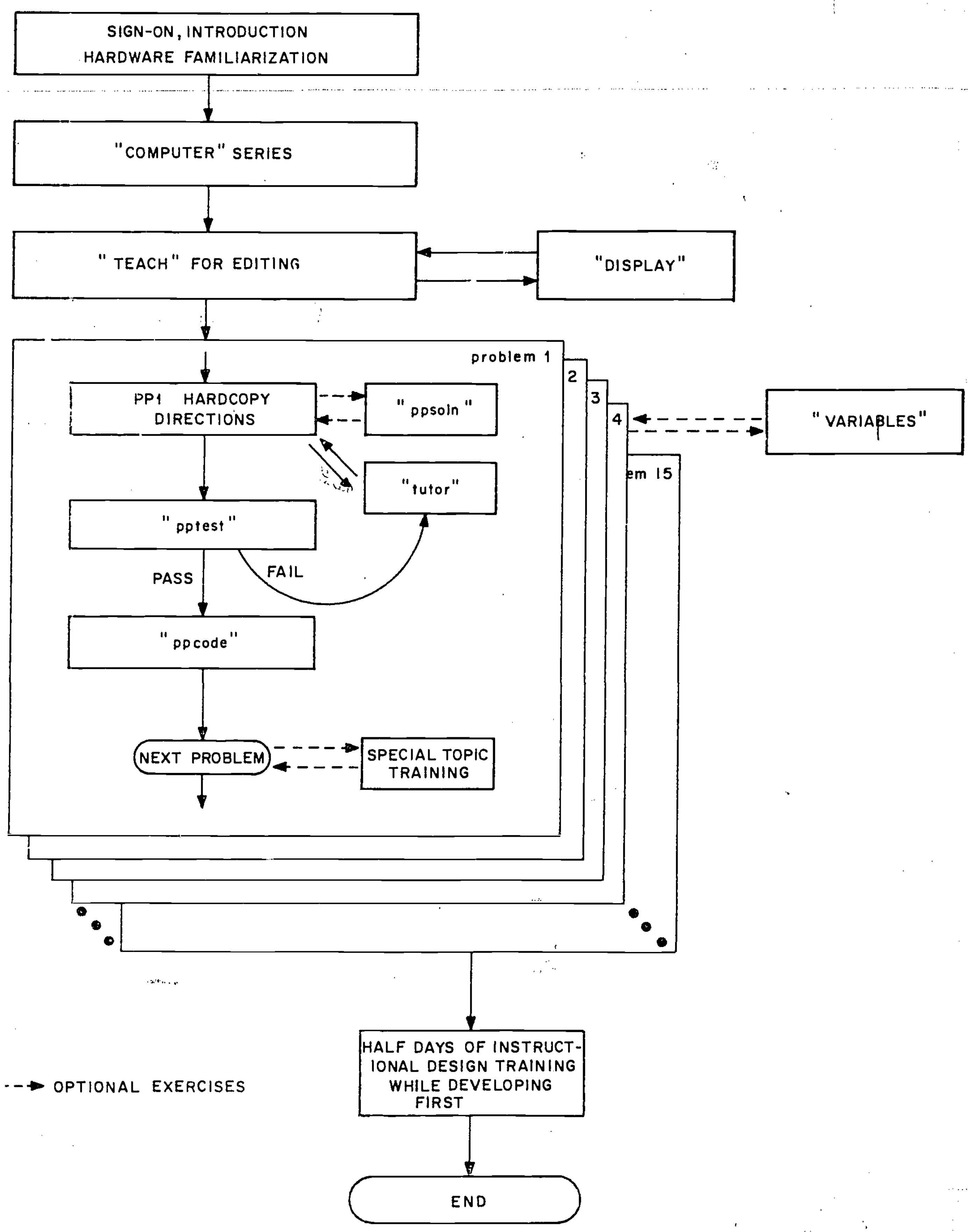
Implementation of the Principles

Principle 1--small segments. Information is introduced in small segments with intervening practice throughout the 15 programming problem exercises (Appendix I). Within each exercise, practice is distributed. The student is able to verify his understanding of the directions and his performance on the exercises by consulting "ppsoln" and "ppcode".

Principle 2--testing. Nearly every exercise is followed by a test. (Each programming problem test or PPT is present on-line in lesson "pptest" and examples from that lesson are in Appendix II of this report.) Examination of the tests demonstrates the problem-solving (i.e., not recall) style of the questions. Where appropriate, multiple forms of the correct answer are accepted (Appendix II, PPT-8, PPT-14). Whenever possible, computer-generated test items are used (PPT-4, PPT-6). In Appendix II, the computer-generated components are indicated by a gray background.

Principle 3--demonstrate a need. The introduction of the -EDIT-

FLOW CHART FOR MTC AUTHOR TRAINING COURSE



key in programming problem 13, part "a," and the introduction of the "Save" option of the editor in programming problem 4, part "i," are two of the more obvious examples of demonstrating a need for a feature or command before teaching it.

Principle 4--intermix topics. The programming problems partially reflect the principle of intermixing teaching on various aspects of the PLATO system, but the actual conduct of the course (not documented here) is a richer source of examples. Instructional design and elementary terminal maintenance are mixed with TUTOR, for example.

Principle 5--self-pacing. With the exception of a few introductory discussions on non-TUTOR topics, the course is self-paced.

Principle 6--training in specialized subtopics. This was implemented as part of class management procedures by assigning authors special topics to study. The selection, assignment, and testing procedures were not automated except that the individual progress data used to make assignments were retrieved from computer records based on performance in "pptest".

Principle 7--interactive reference manual. Each programming problem directed the student to look up the appropriate commands in the "tutor" lesson series. Appendix III attempts to portray the interactive nature of that on-line reference.

Principle 8--new concepts. We found that new authors had a great deal of difficulty with concepts such as looping. They could not see the need for repetitive operations of this kind or relate them to procedures in everyday life. In short, looping seemed a special procedure as mysterious as the circuitry of the computer itself. To introduce some feeling

for the need for looping procedures, we began in programming problem 2, part "f" with a simple animation. Next, problem 4, parts "c" and "i" introduced the idea of incrementing a variable on which the same calculations were repeatedly performed. (The repetition was done by duplicating the coding, rather than by truly looping.) In problem 7, the concept was slowly generalized. The variable -at- from problem 4, part "c" was retrieved and re-used. In the first part of problem 7, the unit was initially re-executed via student-controlled branching, then with author-controlled branching. Each of the three parts of problem 7 (with -next-, -goto-, and -jump-) slowly faded the cues about what the computer is doing internally. By the time the author attempted the -do- loop exercise, problem number 8, he had been exposed to all of the prerequisite concepts for looping, except increment size.

Principle 9--discovery learning. Examples of leaving matters for the author to discover are found in programming problem 1, parts "c-f" and "i," and problem 4, parts "f-g." Persuading authors to adopt a "try it and see" philosophy was taught by example and by precept. Interestingly, there was one author who never overcame his fear of "breaking" the system.

Principle 10--making authors act as students. Lessons "teach", "display", "variables", and "computer" are a start in this direction. However, they lack in-lesson mastery tests and are less sophisticated than is required to fulfill this objective thoroughly.

Principle 11--models of successful authors. This is a topic perhaps best taught by apprenticeship. Unfortunately, the longest any group spent in training at CERL was a few weeks, not long enough to allow time

for an apprenticeship. Though the MTC staff attempted to act in some sense as models for author behavior, we necessarily had a different role, as consultants.

Principle 12--advanced training. We endeavored to provide training on some concepts which were too complex for new authors by a number of procedures described in the section on advanced training (see Appendix IV). In addition, we created lessons such as "execute" and "datacollct" to support training via self-study and MTC-sponsored seminars.

Principle 13--"half-baked" lessons. Although the MTC group is convinced of the value of this technique for TUTOR training, our only major use of this approach was in instructional design training. Given more resources, we would create "half-baked"or "mangled" lessons to be completed and debugged by new authors. Copies of functional, optimized lessons would be implanted with a selected list of errors, difficulties, and weaknesses. Finding and correcting these problems, then comparing the result to the original lesson would allow a new author to encounter and solve problems he would not ordinarily face until well after training was completed.

The first five principles were recognized at the time the course was created. The rest were derived from suggestions, observations, and experimentation which followed.

Development of the Materials

Author and Instructor Backgrounds

The MTC basic TUTOR training package was developed for a specialized audience of new authors. The following conditions were assumed or found to hold for this group:

1. The sites to be established were few in number, seven, and geographically remote from CERL and other PLATO sites.
2. Small groups (4-10) of authors from a single site were to come to CERL for a 2-3 week training course.
3. For most of those authors, the initial training would be the only opportunity to visit CERL and the only extended period devoted to formal training.
4. No CERL staff member or other person knowledgeable about the PLATO system would be employed on-site. Only a limited number of visits to the site by CERL staff would be possible.
5. No staff familiar with CBE, CMI, or other automated education techniques would be available on-site as project employees or consultants.
6. The same CERL staff who trained the site members would make any site visits and would be responsible for consultation.

The above conditions held true for most sites. The actual numbers of sites and authors to be trained was about three times what was expected. The range of author ability and background was substantially larger than we anticipated. A few held computer science degrees, but after two years work with PLATO, several could not meet the training standards designed to be mastered after the two week basic TUTOR course (see appendix VII). The motivational levels of the authors were mixed: some had worked hard to be selected for a PLATO project, others had been chosen from a pool of "rejects" from other tasks.

Table 1 portrays some of the characteristics of the personnel we

trained. It does not reflect the total administrative or support staff at the sites, but rather the staff that received author training.

Table 1

New Author Characteristics

SITE	TOTAL STAFF	NO. OF ADMIN	NO. OF AUTHORS	NO. OF MILITARY AUTHORS	NO. WITH CBE EXP	NO. WITH ID EXP	NO. WITH COMPUTER EXP	AVERAGE AGE
ABERDEEN	10	1	9	5	0	2	1[a]	45
SHEPPARD	12	2	10	10	0	1	3[b]	—
MONMOUTH	10	—	—	—	7[c]	5[d]	3[e]	38
CHANUTE (INITIAL)	11	3	8	4	0	0	2[f]	27
CHANUTE (OTHER)	13	6	7	3	0	2	2[g]	—

Note. The table above reflects the staff trained by the MTC group (basic and/or advanced) at several of the major sites. It does not necessarily reflect the total staff at a site, nor is it a comprehensive list of groups trained by the MTC group. The MTC group trained at CERL or on-site a total of 85 authors. Another 40 ARPA authors and 125 non-ARPA authors were trained by site personnel using materials developed by MTC.

[a] Grad certificate in ADP, left project after four months.

[b] Two who knew several languages, one who knew FORTRAN.

[c] Average of 2.5 years experience.

[d] Average of eight years experience.

[e] From 5-17 years experience. (Note that because some individuals had broad experience, totals for Monmouth exceed the number of authors.)

[f] Each had one college level course.

[g] One year and three years experience, respectively.

Just as the backgrounds of the authors are important for understanding the environment of training, the backgrounds of the MTC instructors are also relevant. Initially, three staff members with PLATO experience ranging from 3-4 years conducted the training for the authors at Chanute, the first MTC site. As the MTC group grew, it added staff without PLATO experience; later, experienced staff could be found.

SITE	TOTAL STAFF	NO. WITH PLATO EXP	AVG YRS EXP[a]	OTHER COMPUTER EXP	AVG YRS EXP	OTHER ID EXP	AVG YRS EXP	AVERAGE AGE
MTC (INITIAL)	5	3	3	2	2	3	2	28
MTC (OTHER)	13	6	2	3	1	4	2	27

[a]Average number of years of PLATO experience possessed at beginning of employment (for staff with previous experience only).

The First Attempt--Eight Statement TUTOR

MTC's initial training exercises were developed around the concept of a "mini-language," Eight Statement TUTOR. TUTOR usage on the PLATO III and IV systems was examined to determine which commands were used most frequently. With this data as a guide, eight often-used commands were selected. Using only these commands, one could construct a complete, if somewhat simplified, lesson. The eight commands included:

1 command used only for organization and naming (-unit-)

2 commands which were usually used together (-at-, -write-)

1 command with a tag form identical to -at- (-arrow-)

2 sets of commands easily and obviously paired (-ok-, -no-; -answer-, -wrong-).

These commands were introduced together in a classroom lecture and required for the first exercise. By grouping the commands as noted

above, only five significantly different kinds of commands needed to be explained. Furthermore, by fitting more and more commands within the structure generated by the basic eight, we felt that all of TUTOR could eventually be learned via a "spiral" path. Each revolution of the spiral--there were three in basic TUTOR--added new judging and display techniques. The commands to be added during the second and third cycles were also chosen based on frequency-of-use data.

The main difficulty in teaching TUTOR this way was that though the number of commands was few, the structures that could be built were complex. Especially problematic was the fact that the implicit branching built into these powerful commands caused some commands to be re-executed or reprocessed two or three times. For those authors not familiar with computers, it was difficult to explain how the computer was rational, orderly, and sequential while explaining the branching implicit in TUTOR. Authors were typically confused by the fact that some commands seemed to function consistently regardless of where they were placed, whereas other commands seemed to function differently depending on where they were placed, i.e., the commands interacted with each other. More than a few authors had the impression that the computer somehow "stood back," scanned all the commands present without regard for their order and then somehow chose the proper ones to execute! Based on a recognition of these misconceptions, MTC created a series of lessons, exercises, and tests to teach even more carefully the exact order of execution. The lessons, however, had only limited success; authors could make simple coding work, often without knowing why, but had substantial difficulty and frustration

trying to add to their knowledge. In order to prevent a great deal of author frustration, a huge manpower effort (high trainer/author ratio) was needed. Commands taught later in the course (for calculation, student-controlled branching, iteration) were not so well understood because their functioning depended on the structures built upon the still-hazy basic eight commands.

After two administrations of the TUTOR course in this form, MTC revived a PLATO III idea which had never been completed and tested.[2] The central concept was to delay introduction of the complexities of answer judging (-arrow-, -ok-, -no-, -answer-, -wrong-) until the very end of the basic course. The rationale for this ordering was to introduce the difficult topic of answer judging after:

1. the concept of orderly, sequential execution was well understood and after

2. the student had experienced success and had built up his confidence by learning some of the easier TUTOR topics.

Coincident, but underestimated, benefits accrued because:

3. fewer commands needed to be introduced initially, thus allowing new authors to "ease into" TUTOR more slowly, and because

4. answer judging was, for most people, easier than calculation and looping, the topics that immediately preceded them in the new, revised order.

Because of the difficulty with calculation and iteration, many authors in the original courses felt that "the farther you go, the harder it gets." By the end of the basic course, their motivation to begin

[2]"Non-arrow TUTOR," R. A. Avner, unpublished paper, 1970.

writing a lesson or to learn new commands was very low. On the other hand, placing the now-easier answer judging following the more difficult TUTOR topic caused authors to gather momentum toward the end of the basic course. Even if authors were bogged down in the calculation and iteration exercises, when they heard from faster colleagues that there was easier material ahead, they were encouraged to keep working.

Simultaneous with the revision of the order of the topics, the portion of the course on editing and controlling the terminal was separated from the first exercise and made a separate activity (lesson "teach"). This modification significantly reduced initial frustration.

It is not possible to make a clear comparison between authors trained via Eight Statement TUTOR and those trained with the revised TUTOR exercises. The Chanute authors remained at CERL for seven weeks because of delays in terminal delivery, a factor that allowed on-going training to remedy deficiencies in the training. Monmouth, the other group trained with Eight Statement TUTOR, had a great deal of prior CBE background. Subjectively, however, we found that the revised ordering enormously improved the speed, efficiency and attitude of new authors.

Beyond Eight Statement TUTOR--A Description

In the initial version of the TUTOR training materials, the MTC group used off-line copies of the on-line TUTOR reference lesson "aids" as background material, introduced most new information via lecture, gave oral tests for each exercise, and obtained written critiques following each exercise. Though thorough, this process required a great deal of labor. As time progressed, MTC had more non-training

responsibilities and hence wished to reduce the manpower allocated to training. Furthermore, we had accumulated sufficient data both to improve the course and to decrease the trainer/author ratio.

Authors found the "aids" materials (which were not written to teach TUTOR) too detailed and generalized. Therefore MTC, with the aid of other CERL staff, created a series of reference lessons ("tutor") for novice authors. These on-line references were written as a simplified version of "aids" but more importantly, were interactive--that is, the new author could in many cases "see" some of the hidden steps in the execution of a command (e.g., -help-) or input sample commands and observe their effect (e.g., -bump-). The "tutor" series contains a subset of the TUTOR language (43 commands) and descriptions of six concepts applicable to many commands (e.g., variables).

The oral tests were first changed into paper-and-pencil tests and later were put on-line (lesson "pptest", for programming problem test). Computer generated and graded tests enhanced test integrity and provided a mechanism for giving the author the "answer" to his programming exercise without permitting "cheating" (see below). For affective objectives, we found no techniques as efficient and thorough as those used for testing knowledge about commands; subjective ratings and comments were recorded.

Two more series of lessons were created to define the author's job more precisely and to catch subtle errors. All of the programming problems (exercises described on paper) require the student to create some sort of program. Authors sometimes found it difficult to understand what was to be done without first seeing it on the terminal. Hence, a

series of lessons "ppsoln" (programming problem solutions) was created to show the student his "target" as it should execute in student mode. Also, because of the flexibility of TUTOR, it was possible for an author to create a program that seemed in accord with the instructions when, in fact, he had overlooked a detail. To remedy this, an author was allowed, after he had successfully completed the "pptest" for an exercise, to view a "properly" coded solution (series "ppcode").

The "programming problems", the paper directions for the exercises, were updated, elaborated, and clarified during each revision. In addition, many more questions were added, and the oral lectures were replaced with written explanations. These paper lectures are terse, simplified explanations which rely heavily on examples. Some lecture materials were put into the form of a PLATO lesson ("teach", "display", "variables", "computer") to give the author insight into a student's problems in learning from the PLATO system.

Because changes in command names and formats required updating of the training materials each time the course was offered, there were ample opportunities for making more substantive changes to the exercises, to lesson "tutor", and to all materials in the training course.

Advanced TUTOR Training

The basic TUTOR course was designed to be the only formal instruction ever to be presented to the ARPA/PLATO authors. However, based on our perception of the style and quality of the lessons being produced and in accord with our principle of teaching a topic when an author has a need and context for it, we decided to supplement basic TUTOR training with advanced TUTOR training to be given on-site.

Although a substantial amount of training was done on-site (any site visit during the first 6-8 months after training included one or more advanced TUTOR sessions), we found authors generally resisted formal instruction (including training during MTC site visits) once they had left CERL. If pressed, they would rather listen to a 1-3 hour lecture on advanced topics or new features than work even a few paper "exercises." But in the main, they would rather not endure any formal instruction. Once in the field, they were under considerable time pressure and therefore relegated training to a lower status.

Documented Drivers

Under these circumstances and in an attempt to bolster the skills of seemingly underprepared programmers, we created two "documented drivers," (pre-programmed routines accompanied by extensive documentation).[3] The routines can be easily incorporated into a lesson to provide standard, but hard-to-program, practice or testing paradigms. These carefully described drivers were intended to allow an author to learn about advanced

---

[3] See Appendix IV for an example.

features and techniques in an actual working context; presumably, since authors felt training exercises were "time out" from their production time, they might be more willing to spend time to understand drivers which would be useful in their lessons.

The "documented drivers" sought to meet four objectives. First, each was a response to an author's specific request for programming help, modified to be more general. Second, the drivers were to provide unsophisticated authors with routines more powerful than those they otherwise might employ, but containing features they would program if they knew how (e.g., providing an "input now, judge later" matching quiz which allows the student to change answers, provides student feedback, and prevents impossible answers--in general, a "never-fail" matching routine). Third, they were a vehicle for teaching advanced coding. Hence, rather than providing a "black box" routine or even including a standard level of documentation, the drivers were explained in great detail. For example, one provides a description of "segmented" variables, a topic not taught in the basic TUTOR training course.

A fourth objective, not related to TUTOR training, was to provide instructional design guidelines related to use of specific testing formats. For example, the matching driver contained sample directions to the student and suggestions for parallelism between alternatives, number of alternatives vs. number of questions. We had previously noted that such considerations had been overlooked by new authors.

Several forms of a hardcopy questionnaire were used to assess the usefulness and effectiveness of this approach. In addition to providing

some useful feedback about specific changes and additions to the documented driver, the respondents answered questions which indicated: 1) they would find additional documented drivers helpful; 2) they found the documented driver about as useful as several other manuals, references or aids they were using or had used; 3) they all felt documented drivers were most useful for authors just starting to develop their own lessons (as opposed to "new authors" or "authors who had written and student-tested several lessons"). Interestingly, the only uses we know about were by "authors who had written and student-tested several lessons," but no respondents selected that choice.

Though it is not possible to judge how much information was transmitted in this fashion, an examination of 35 Chanute lessons shows extensive use of a matching driver: seven lessons use the MTC matching driver, and 26 lessons use a version of that driver as modified by Chanute staff. Only one use of the randomized drill diagram was found in the Chanute lessons (though it was a Chanute author who requested this routine be written). Fort Monmouth was the only site other than Chanute which employed these "documented drivers." We discovered later that at some other sites, the materials were kept by the director and not circulated to the authors. By that time, however, most authors had mastered the concepts the drivers sought to teach. Because the need had decreased and because we lacked indications that this approach was especially successful, later drivers created by the MTC group did not contain the extensive documentation included solely for the purpose of TUTOR training.

Calculation Training/Retraining

Calculation was the topic treated most regularly during on-site

visits. Not only was it the topic ARPA authors found most difficult during the MTC course, it was also the one which they had greatest difficulty mastering during the follow-up phase. Although there are many facets of TUTOR which are quite unrelated to calculation, an author's ability with calculations often serves as a useful measure of his overall programming ability. Unfortunately, only a few of the ARPA authors (10-20%) ever gained a really good mastery of calculation. Incomplete understanding necessarily hindered or prevented the development of certain complex teaching strategies. In quite a number of cases, the MTC group provided coding for complex programming tasks; however, it is clear that MTC's efforts could not, in any case, completely compensate for low comprehension of this topic. In fact, at several sites we have found instances where authors simplified or discarded plans because their coding experience and skills were too weak to do the programming needed for the ideas they envisioned.

We feel that most authors failed to master this topic because they had backgrounds weak in mathematical problem-solving. Authors with mathematical or scientific backgrounds had few difficulties, whereas authors with other subject matter backgrounds fared worse, in general. Regrettably, some authors of two to four years experience still do not use or understand expressions with indexed variables[4], a topic taught in the basic TUTOR course.[5]

---

[4]For example, the indexed variable v(v1) in the expression "calc v(v1)⇐ 4" indicates that the value of v1 should be used to calculate which variable is set equal to 4. If v1 equals 10, variable 10 (v10) is set equal to 4.

[5]See Appendix VI for a survey of use of indexed variables.

In our training we found a significant initial stumbling block to be the concept of a "variable." We found four levels of understanding of this concept.

1. The author understands that "time 15" causes a 15 second wait and that the name "delay" can be attached to a constant whose value is 15.

2. He understands that "delay" could instead be a variable, say v1, whose value is 15. Few authors have difficulty understanding that adding or subtracting from "delay" varies the length of the timed wait.

3. At the next level, new authors are able to properly handle the problem "What if 'delay' contained the time in minutes, rather than in seconds?" and other problems with simple arithmetic or algebraic solutions. Most authors can type "time delay × 60 with confidence.

4. Authors do not easily generalize these results to commands like -at 1014-. Knowing that this means 10th line, 14th space, they "see" 1014 as two numbers. When they use an -at place- (where "place" = v2 = 1014), they cannot describe how to add or subtract from "place" in order to vary the screen location. In order to move the position of the -at- down one line, some could be expected to increase "place" by 64 (the number of spaces in a line), rather than by 100. They might even claim that because there are "really" two numbers in "1014," it is impossible to adjust "place" arithmetically to indicate a position one line down. Although this quandary can often be resolved by pointing out the new position and the tag of a corresponding -at- command (11th line, 14th space or "1114"), some authors still have difficulty realizing that values held in variables are indistinguishable from those typed explicitly (i.e., that "1014," placed in a variable which is used as the tag of an -at-, behaves exactly as if the author had typed it directly). Likewise, the use of logical operators in conditional commands (e.g., "jump count > 5, true, false") is understood more easily than the use of the arithmetic values of logical expressions (e.g., "time (count > 5 + 1"). We estimate a quarter or more of the ARPA curriculum developers did not master the fourth level, the most generalized concept of a variable. Programming problems 4, 7, and 8 (and the associated tests) attempt to teach this concept.

The formats of variables also cause confusion. Authors introduced to both "v" and "n" (floating point and fixed point) variables often

believe that there are two sets of 150 student variables. Likewise, alphanumeric format is often not understood thoroughly enough so that alpha information can be manipulated. Many authors limit their operations to storing and retrieving alpha information.

Data Collection

In the course of working with authors who were testing their lessons, we found that they were not extracting all the information they might from the student datafiles. Authors typically looked only at the progress of individual students in a one-at-a-time mode. Therefore, MTC wrote a pseudo-lesson "datacollct" to generate student data of all types. A datafile was filled with data generated by simulated "bright" and "slow" students in easy and hard sections of the lesson. There were miskeyed questions, too-difficult questions, execution errors--in general, all of the kinds of problems that can be found and diagnosed via analysis of student datafiles. We intended to create a series of exercises to give authors practice analyzing datafiles, but this seemed to be a cure for a disease which the victims had not yet recognized. Hence, this datafile analysis technique was explored, but never fully implemented; this approach seems more acceptable for teaching new authors than for upgrading the skills of semi-experienced authors. Unfortunately, at the time it was developed, few new authors remained to be trained. Therefore, only one addition was made to the basic lesson described above: a set of routines which demonstrated how student data could be stored, sorted, and retrieved from datafiles.

<u>Other Advanced Training</u>

To the credit of the designers of the software, training in the areas of graphing, matrices, character sets, linesets, and advanced graphics can be included either at the time of basic TUTOR training or can be learned by self-instruction. The directions are clear, the options well-defined, and the results predictable. We have never produced exercises on these topics and do not ever see a need for them.

Unresolved Teaching Dilemmas

As in most teaching situations, problems arise which are not solved; some of these situations offer several alternatives, each with unpleasant results--dilemmas. Many of these dilemmas involve a situation in which the new author lacks experience. He may possess incorrect intuitions or misconceptions which are too strong to be removed by lectures, reading, or any exercise that can be included in a training program. The new author must learn by experience. In other cases, the difficulties are not easily perceived by the author because they cause problems for someone else (students, administrators, etc.) or because the problems appear only at a future date. For many such dilemmas there is a fruitful alternative; if the supervisor of the new author is experienced, he can firmly insist that procedures (e.g., documentation) be followed. When a manager experienced in CBE development takes charge, many of the dilemmas listed below vanish. Unfortunately, CBE is a new field and managers skilled in CBE development are in short supply.

Dilemma 1--How to Deliver Advanced Training

When authors in the field are asked what kind of further training, aid, and support they want (in advanced TUTOR, instructional design, etc.), they respond that they would rather have a review[6] (of the code or instruction design) which points out specific items for improvement rather than mini-courses, workshops or exercises. They point out that the specific

[6] A critique. A more detailed explanation is found in Lesson Review, L. Francis, M. Goldstein, and E. Sweeney, MTC Report no. 3, CERL, December , 1975.

critique is a more efficient use of their time, telling them nothing about areas they are handling satisfactorily, and contributing to production items rather than practice examples. This is logical; however, we have seen repeatedly that reviews produce little transfer of learning. In a review, many authors view general comments as unsatisfactory (e.g., "units are of unwieldly length, feedback is sparse," etc.), yet are unable to generalize specific comments to other cases and hence, repeat mistakes. Our hypothesis, untested, but based on substantial interaction with authors during lesson reviews, is that the cause of this transfer failure is the author's relative inability to discriminate between satisfactory and unsatisfactory material. Improving his discrimination, however, would probably require drills or exercises to a mastery-plus level--just the sort of activity authors seem to wish to avoid.

Dilemma 2--When to Teach Documentation

Documentation, both on-line and off-line; flowcharting; and "software engineering" (the use of coding standards, meaningful block and define names, etc.) are frankly not needed for the very elementary programs new authors create. For that reason it is difficult to convince a new author to document thoroughly. The extra time needed seems certainly a waste. Few new authors need to modify someone else's coding and they may not need to look back on their own for 6-12 months. Unfortunately, after that amount of time, bad habits (poor documentation) have already been established and as much as one year's worth of work is poorly documented. Some typical bad habits are worse than merely confusing. Failure to set

aside a group of variables which can be used for special purposes across a series of lessons can require an inordinate amount of work later, or, more likely, result in abandonment of a review management strategy which requires those variables. In conclusion, it is difficult to persuade a new author to use documentation, without more forceful incentives than mere encouragement. (However, see principle number 13.)

Dilemma 3--Taxonomies in TUTOR training

A general problem in all of education afflicts, not surprisingly, TUTOR training. It is much easier to teach and test the mechanics of performing a function rather than the judgment to know <u>when</u> to perform it. For example, we regularly discovered new authors who had completely filled a block and who were stymied about how to acquire more space. Because of the training the author had received, we knew that these authors could create new blocks; however, they failed to realize either 1) the nature of their problem or 2) that a task they could perform was the solution. This training problem can be remedied by increased proctoring, and hence costs, or by creating a "no-more-room" situation in a controlled environment as part of an exercise. For example, the author can be instructed to add ten words of programming to a block which has room for five words. Appropriately written instructions can capture the author at this point of keen interest, yet before frustration occurs. We have found this technique much more effective than providing in advance a description of the problem and the solution. In many cases, single-trial learning has been accomplished.

Unfortunately, these sorts of situations do not frequent' ccur

within the control of the TUTOR instructor. Often appropriate situations cannot easily be created and hence occur only for a few lucky individuals. In other cases, a potential learning situation may occur, but pass by unnoticed and unexploited because of the absence of a TUTOR instructor who can turn it into a learning experience.

Since one of the important tasks for both instructional materials and instructors is the creation of learning incidents, the PLATO system can and should be used to simulate problems while simultaneously controlling the situation. A simulation was used for teaching elementary editing and command usage as early as 1970 (on the PLATO III system), although that implementation was unsuccessful. There, most authors could complete the simulation successfully, but could not solve actual problems when they arose. It is not clear if this implementation failed because the authors were too strongly prompted during the simulation, because the program failed to separate the instructions and feedback given in the simulation from information and displays provided in the standard editor, or because the authors learned the tasks only mechanically with no real understanding of what they were doing. It does seem that it failed to take advantage of the capability to generate learning incidents. A more recent simulation ("introedit"[7]) was used with considerably more success to teach editing to some of the last PLATO/ARPA authors to be trained by MTC. The mixed success of PLATO simulations is difficult to explain further. Differences in small details of a lesson's display layout,

---

[7]by Silas Warner, Indiana University, 1975.

directions, or student control options may actually overpower the seemingly larger effects of lesson strategy and pedagogical technique. For example, the MTC lesson "teach" which instructs the author in basic editing has been used successfully with few problems for training many ARPA/PLATO authors. A modified version of this lesson was used by the PLATO Services Organization (PSO) staff until they concluded that authors could not relate the actions performed in the modified "teach" to the solution of typical problems. It is not clear whether the modifications to "teach" or the mode of use accounts for the different perceptions of effectiveness. Unfortunately, lack of documentation of the PSO modifications and style of use precludes analysis.

Dilemma 4--How NOT to Pass on Experience

The MTC group discovered that possessing lesson development experience and using it to predict problems and outcomes is not always an effective way to prevent those problems. It may, in fact, be deleterious.

Based on substantial prior experience, even at the time of the Chanute training, MTC felt strongly that each author should choose a very short subject for his first lesson. We felt the lesson should be written as fast as was reasonably possible, tested on students, and "put on the shelf" for six months, to be revised or discarded at that time. It should either not be revised at all or should be revised only once, the latter only if execution errors prevented collection of student data. We predicted most first-try lessons would be discarded eventually. We opposed revision and predicted eventual deletion based on the premise that a new author learns so much new information in the early stage of his

career that after six months he would want to rewrite the lesson whether or not it had been revised previously. Furthermore, using the time of a novice author to revise a lesson would mean the lesson would be completed only very slowly. We felt the amount a new author could learn by revising his first lesson, at this stage of authoring, would be less than he could gain by beginning a new lesson. Six months later the same author, with expanded coding and instructional design experience, could revise or rewrite the initial lesson in a very short time.

It is most important for an author with little experience to go through the lesson development process all the way (especially running students and analyzing student data) before attempting to upgrade his lesson. The ways in which he would modify it after going through the whole developmental cycle are typically different than the way he would change it if he revised it before obtaining student response data. Hence, it is important to require the author to complete his first lesson development cycle just as soon as possible.

MTC staff discussed with the Chanute authors the pitfalls of developing their first lesson, our predictions about it, and our recommendations. Our intent was to give them a glimpse at the future and to provide them with an optimal path. Unfortunately, they mistook the meaning of the information. They felt that, armed with the knowledge that a typical author discards his first lesson, they could skip a step. The authors felt they COULD write a first lesson good enough that it need not be discarded.[8] This conclusion,

[8] We found later that authors seem to abhor deleting anything. Ego involvement is high where large amounts of creative effort are expended. Even so, we have been surprised by an author's view of the sacrosanctity of the computer word; many authors refuse to delete the coding generated from the practice exercises of the basic TUTOR course until months afterward.

coupled with the normal novice author's tendency to "hide" a lesson until it is polished, produced results exactly opposite to those we desired and anticipated. Encouragement by MTC personnel to finish the first lesson and begin a second was cause for ill will towards MTC. The new authors felt the "first lesson will be scrapped" philosophy was condescending: "Ever since the training course at CERL, we were treated as novices. We are often told that we won't (can't) produce a good lesson the first time around."[9]

Consequently, the authors spent a great deal of time polishing and repolishing lessons which had not yet been "rough sanded." In some cases they revised their lessons on a nearly daily basis to incorporate new commands or strategies they had learned.

---

[9] Comment from semi-annual reports written by each author (September 72-April 73) furnished by Capt. J. Green, Chanute AFB.

Teaching TUTOR in an Environment of an Evolving Language

One unusual aspect of the MTC TUTOR training course was that it was conducted in the environment of a constantly evolving language. Several effects on the authors, instructors, and training materials were noted.

Surprisingly perhaps, the effect of the changes on the training was not especially deleterious. In the course of a 2-3 week training session only a few changes typically took place. These generally involved the introduction of new commands or addition of new features to old commands. Because TUTOR had been in development for some time before the first authors arrived for training, the basic commands in TUTOR had settled into comparatively final form. Also, in most cases, the MTC group was warned about upcoming changes and could take steps to minimize their impact. In one case, the TUTOR editor was modified to treat ARPA/PLATO authors specially until our training materials could be updated.

Most of the changes have made the job of teaching easier. The most radical (from the new author's viewpoint) changes that occurred were those that affected how one moved about in the TUTOR editor. In 1972 and early 1973 this was a very complicated matter. Examination of old MTC documents shows at least four distinctly different "maps" of the "pages" of the PLATO software (author mode, student mode, security code changing, etc.). The current organization of the editor is so straightforward that we no longer supply "maps" with our training--we used to spend 1-2 hours on the subject when there were fewer options available from each page. That is, there were formerly more levels of "depth" with fewer options at each level. The current organization is "shallower" with consequently fewer keys needed to

gain access to the same information. Similarly, selection of options via function keys (-LAB-, -DATA1-, etc.) has largely been replaced with selection via single letters automatically judged (without pressing -NEXT-). Options are keyed to the name of the operation or information desired (e.g., "S" means "security code"). Lastly, the current structure, when drawn as a "tree" is much "cleaner." There are fewer crossovers between "branches" now--fewer special cases.

In another case, the level of TUTOR training we aimed for was significantly changed when the "ID/SD" interactive graphics editor was introduced. This feature made the design of complex graphics so quick and easy, it both solved and created problems. Rather than teaching most display commands in detail, we shifted to teaching each author only enough so that he could generally relate the system-generated commands to his display. This proved to be a much simpler task than teaching him enough TUTOR to generate the coding himself. The only problem we met was that creating displays became so much fun that we had to place a time limit on one of the programming exercises. Otherwise, authors would have spent the day drawing pictures.

Changes to commands followed somewhat similar lines. They tended to simplify common uses or expand sophisticated use. In nearly every case, the lessons already written were automatically converted to the new form of the command. The MTC training materials, however, had to be regularly updated.

The introduction of new commands that were easier to learn and understand often meant that less efficient or straightforward commands were allowed to remain in the system to prevent old lessons from becoming obsolete. New authors had no problems unlearning old forms; they had never

learned them. However, the MTC instructors were necessarily staff of long experience. Although there was never resistance to the improved commands, the old habits sometimes died slowly.

Evaluation

The progress of the authors and the appropriateness and quality of the training materials were monitored continuously during development and use by means of rating forms, paper or on-line quizzes, and, during the first year, a final overall test. Because materials were under continual revision, these internally gathered data served mainly for formative evaluation. Evaluation of the finished product is best measured by reports of users outside of MTC during this period.

The Chicago community college authors associated with the NSF Demonstration Project were taught the fundamentals of TUTOR in a series of courses taught by CERL staff. In the Spring of 1973, twenty-four authors from Chicago spent a marathon weekend in Urbana learning TUTOR. Rather than receiving instruction from the Community College Coordinator (a senior educator with CBE and instructional design experience who had carried out previous author training), the authors were trained using a shortened version of the MTC TUTOR course. The instructors were selected MTC, PEER, and other CERL staff.

> The general consensus was that the [semester-long] spring course was far superior to the fall course in every respect. One reason given was that early in the course the class spent a weekend at CERL with full access to the terminals and met with the subject matter area coordinators. They returned from that weekend with a good up-to-date understanding of the newest TUTOR commands and much practical experience. "Our group had better training. We learned the powerful commands--mode erase, variables, etc."[10]

---

[10] "PLATO comes to the Community College," E. R. House and C. L. Gjerde, Center for Instructional Research and Curriculum Evaluation, University of Illinois (1973), pp. 35-36, [quote from pseudo-named author].

An evaluation of the weekend workshop just as it ended also suggested the participants found it highly valuable. Asked, "Would you recommend that a friend with the same background and interests as yourself take a workshop such as this," the 20 people completing a questionnaire responded:[11]

Definitely Yes	12
Probably	6
Uncertain	1
Probably Not	0
Definitely Not	0
No Response	1

More recently, a Federal Aviation Administration (FAA) Academy staff member tasked with training new authors reported that in addition to using other MTC materials, the new authors,

> have been in and out of "introtutor" [see Appendix V] and have spent a lot of time in "pptest" [and] . . . "variables"; it appears that the MTC stuff is much more valuable in that it provides constant practice and testing.[12]

The MTC package was adopted as the training method for staff of the PLATO Medical Project (a 60-terminal group).

Approximately 20 students in a University of Illinois CBE course have been trained using these materials, and the materials were adapted for classroom (i.e., off-line) training by a Human Resources Research

---

[11] "PLATO Evaluation Report--TUTOR workshop 26-28 January 1973," R. A. Avner, Computer-Based Education Research Lab, University of Illinois (1973).

[12] C. N. Burson, private communication, June 22, 1976. Oklahoma City, Oklahoma.

Organization (HumRRO) staff member in order to instruct personnel in that organization.

Because of the easy access to document copying machines, we do not have an accurate estimate of the extent of the distribution of MTC materials. We do not provide multiple copies for non-ARPA users, nor does any other agency, including CERL. In fact, the existence of the materials is not apparent to a new non-ARPA user; there is no reference to them in any system-supported lessons, such as "aids". Nevertheless, lessons for which we record users show a steady or growing use. Lesson "tutor", is now accessed about 375 times per month, for example.

Appendix I

The Programming Problems

to accompany the

MTC Author Training Course

The following exercises are part of a manual provided to new authors taking the MTC Author Training Course. It is assumed that the author is already familiar with use of PLATO as a student. Prior to seeing the first exercise, the new author is shown a course outline, given a list of references, and provided with an orientation to the course.

Instructions for Learning to Edit

You have been assigned a lesson called "teach". Ordinarily, when you get a new lesson it is empty. This time, however, we have filled your lesson space with instructions about how to maneuver in author mode.

Although this lesson file is a standard one, you will use it in an unusual way for the first hour or two. You will operate in author mode most of the time. Thus your interaction with the computer will be different than you experienced when you tried lessons as a student.

At the same time, you will not be behaving exactly like an author since you will not be programming very much. Once you get used to moving around in author mode, you will delete the material in your lesson and begin to learn about writing your own code in the TUTOR language.

Note: While you are in author mode you will get few, if any, "ok" or "no" judgments by the computer and few hints what to do if something should go wrong.

Right now sign into your lesson in author mode. On the screen that lists the LESSON and BLOCK press the "b" key and don't return to this direction sheet until told to do so by the instructions in your lesson.

Destroying and Creating Blocks

Go to the block display for your lesson. Fill in the information below in the column labeled "now".

	Now	After
Block d, part 1 has what name?	____________	____________
Block e, part 1 has what name?	____________	____________
Block f, part 2 has what name?	____________	____________
Block g, part 2 has what name?	____________	____________

PLATO automatically deletes a block when there is nothing in it. Go into block "two" and delete 100 lines. This is more lines than there are in block "two," but other blocks won't be affected. When you return to the block display page, you will notice that block "two" has disappeared.

Check the table of information you filled out above to see what blocks have what names and fill in the column labeled "after."

An upper case letter (say, C) typed on the block display page is interpreted as "Insert a new block after the block labeled C." Begin a new block called "start": (1) Put it after the block named "six" by typing "F" (upper case). (2) Ask for a "normal" TUTOR block; name it "sixb." (3) Insert your name in this block, using the regular insertion directive. (This insertion is necessary to keep PLATO from deleting the block because it is empty). (4) Return to the block display page and compare this page with the table of information you filled out above.

Delete all material from all blocks including those that say "Stay out." Press -DATA- on the block page and fill out all lines possible.

Programming Problem 1: Displaying Text via Write

The objective of PP1 is to familiarize you with (1) the -write- command, (2) the character grid system, and (3) the methods used to insert and delete lines of code from your lesson file. When Inserting TUTOR statements, first type the command, press the -TAB- key, and then type the tag. To continue Inserting TUTOR statements, press -NEXT-; to return to the Line Display page, press -BACK-. Remember while viewing the Line Display page you may use the following directives to control TUTOR.

f - Forward (up)	i - Insert
b - Backward (down)	d - Delete

a) Copy the following TUTOR statements into your lesson file in author mode.

Command	Tag
unit	pp1
at	1ø1
write	ABC
at	162
write	DEF
at	32ø1
write	GHI

b) Look at this version of your program in student mode by holding the -SHIFT- key down while pressing the -STOP- key (which is abbreviated: -STOP1-). Notice the screen position of the letters A, F, and G.

Questions:

1. What is the screen position of each letter: A ________, D ________, F ________, G ________.
2. ____________ How many lines of information may be written on the plasma panel?
3. ____________ How many characters may be written across a single line?

c) Add the following TUTOR statements to the end of your lesson file.

Command	Tag
at	???
write	XX

Calculate the correct line and character coordinates so that the pair of X's will be written at the center on the bottom line of the screen in student mode. Replace the question marks with this number.

NOTE: If you are not sure that you have coded this problem correctly, look at lesson "ppsoln" to view a working solution.

d) Look at this new version of your program in student mode. If the placement of the two X's is not in the middle of the line, change the character position of the -at- command so that the two X's are exactly in the center of line 32.

e) Add the following TUTOR statements to the end of your lesson file.

Command	Tag
at	2741
write	This line requires 43 spaces on the screen.

f) Look at this new version of your program in student mode. Observe what happens when the writing of an individual line extends beyond the right boundary of the screen. This effect is called "wrap-around."

Question:

Given the -at- command of part e, how many characters may be written before wrap-around occurs? ________________________

g) Add the following TUTOR statements to the end of your lesson file.

Command	Tag
at	1824
write	This is an example of a -write- command that has a tag consisting of more than one line.

h) Look at this new version of your program in student mode.

Question:

The tag of the -at- command contains two pieces of information: the line and character position. In part g above, the tag of the -at- command tells us that the writing which follows will start at

line position __________________ and

character position ________________.

i) Add the following TUTOR statements to the end of your lesson file.

Command	Tag
at	1Ø1Ø
write	Ø123456789
write	abc...xyz
write	Ø123456789

j) Look at this new version of your program in student mode.

Question:

What are the differences in the way that PLATO executes the TUTOR statements in part g versus part i?

____

____

____

k) The tag of a -write- command may contain a maximum of 112 key presses. Many of the characters that you will be using in -write- statements (i.e., capital letters) require two or more key presses to produce one displayed character. Therefore, the tag of a -write- command in author mode containing capitals, cannot contain 112 displayed letters.

The following line should be displayed on line 5, character position 3. The TUTOR statements should again be added to the end of your lesson file.

Display in capital letters on <u>one</u> line exactly (with underlining):*

LEARNING <u>TUTOR</u> IS SUPERCALIFRAGILISTICEXPIALIDOCIOUS!

l) Now run lesson "pptest" in student mode and take the test for PP1.

*Underscore character is -SHIFT- 6. To backspace once, press -SHIFT-SPACE-.

Programming Problem 2: Drawings

Now it is time for you to create your own code. You will be given the objective and your task is to write the code to meet the objective. In PP2, you will draw figures, selectively erase a character, and do a simple animation.

Use of the following commands will help solve this problem:

unit	write	pause
at	draw	erase

If you are not familiar with these commands, you will find a description of their use in the lesson "tutor". Remember, sign into lesson "tutor" from the author mode page by pressing -DATA-.

a) To help you get started, create a new -unit pp2strt-. Insert the following TUTOR statement after the -unit- command.

```
draw     2020;2530;2040;2020
```

What type figure will be drawn by the above statement? ________________

You are not going to be told when to look at a section of code that has been written as in PP1. When you wish to try out any material as a student, just hold the -SHIFT- key down and press key -STOP- (i.e., -STOP1-). You will then be placed into student mode at the beginning of your lesson. (PP1). To view PP2, just press the -NEXT- key.

b) Write a statement of your own design consisting of two short lines. Display the beginning of this message at 510. Using the -draw- command, draw a rectangle around the message.

c) Display a capital letter inside the triangle, and draw a line from any corner of the triangle to any corner of the rectangle.

d) Suppose we have programmed some type of game and it becomes necessary to erase only the capital letter in the triangle. Add the necessary code which will erase only the capital letter.

e) The capital letter was written and erased from the screen in a very rapid sequence. The objective of this part is to delay the erasing of the capital letter for 2 seconds.

f) We now know about the elements necessary for very simple animations: -write-, -draw-, -pause-, -erase-, etc.

Objective: Start a new -unit pp2move-. Write the code necessary to "move" a character in an animated fashion a few spaces in any direction.

g) Run lesson "pptest" for the test PP2.

Programming Problem 3: The Grid System

The purpose of PP3 is to acquaint you with the fine grid system of the display panel and to practice the directives "id" and "sd" used for creating displays. If you are not familiar with the fine grid system, see one of the following lessons: "tutor" or "aids". Use of the following commands will help solve this programming problem.

unit draw circle

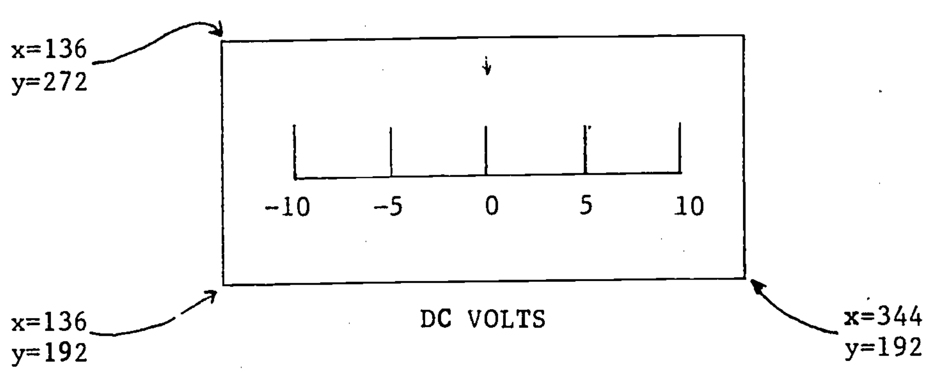
at write

a) Previously we learned the editing directives "b", "f", "d", and "i" for "back", "forward", "delete", and "insert", respectively. Much like "i" which allows you to insert commands in your TUTOR code, "id" allows you to <u>i</u>nsert a <u>d</u>isplay. Typing "id6" will insert the display after line 6. Press -HELP- to see the options available for creating displays while in the "id" mode. In the lower right hand corner will be the words "gross grid" and the numbers refer to where the cursor (the "+" which moves on the screen) is positioned in both fine and gross (character) grids. After drawing a display, press -BACK- to insert the display commands in your TUTOR code.

b) If you had partly constructed a display using the "id" directive or you would like to see a series of display commands, you can use the <u>s</u>ee <u>d</u>raw ("sd") directive to regenerate the display. First, move the first line to be displayed to the top of the screen and then type "sd7" (or whatever number of lines) to see the first to the seventh line displayed. At this point you can continue your display.

Start a new -unit pp3-. Using the fine grid system and the "id" and "sd" directives, draw the following meter face. The important thing is to understand the relationship between the character and point grid systems. Do not spend more than <u>one</u> hour doing this part of PP3.

Note: The arrow shown on the meter face may be made by using <u>existing</u> characters. Ask the proctor how this arrow is displayed.



c) Draw the smallest possible circle you can around the box above. Its radius is ______ dots.

Remember to look in lesson "ppsoln" for solutions to programming problems.

d) Take PP3 test in pptest.

Programming Problem 4: TUTOR Variables

In PP4 you will study TUTOR variables, how to set, change, rename, and display them. For background information, read the course handout entitled "Variables" and the "Variables" topic in the lesson "tutor". The following commands will help solve this problem.

```
unit        define      calc

at          showt
```

Try lesson "variables" in student mode.

a) Start a new -unit pp4set-. First set TUTOR variable v1←1632 and then -showt- (display) the contents of the variable in the center of the screen.

b) Set v2←1010 and display at character grid position 1010 by using -at v2- and a -showt- for the contents of v2.

c) Add 200 to the current value of v2 and again use an -at v2- to -showt- the contents two lines below the number in part b.

d) At times it is useful (e.g., for readability of your TUTOR code by yourself or others) to rename a variable. This is done using the -define-.

   Example: define pos=v2

   The name is limited to 7 characters and must start with a letter. Caution: the -define- must appear before the first use of the new name. Add the above -define- to your lesson and the below code.

```
calc    pos←pos+20
at      pos
showt   pos
```

e) -define- variables for salary, taxrate and tax. Set salary ← 10000, taxrate ← .21. Multiply salary and taxrate and place it in tax. Write a "$" -at- 3005. -showt- the contents of tax after the "$" without using another -at-.

f) Set v4 ← -1234.567 and display with a -showt-. Did you expect the results? Change the tag by adding an appropriate format. Replace the -showt- with -show-.

g) Have the computer calculate the following expression:

   v2 ← 5x6/3x7+5

   NOTE: Display the contents of v2. Study the result of this evaluation and determine the order by which the different operators are executed.

Does PLATO multiply or divide first? ____________________________

h) Set variables v56, v69, v7Ø, v71, v72 equal to zero. (This may be done with a single -calc- statement.) Display the contents of v56, v69, v7Ø, v71, v72 to check.

The following problem is optional; check with your proctor before beginning it.

i) It is possible to reference a variable (e.g., v5) indirectly as shown by the following example.

```
calc    v1Ø ← 5
calc    v(v1Ø) ← 3.14159
```

This will put 3.14159 in v5. Type the above code and add a -show- to see the value of v5.

The purpose of this part is to generate a simple table in which you calculate one column.

Start a new -unit pp4tab-. We will create a table for inflation assuming a 5% rise in cost per year. Assume bread costs 25¢ a loaf in 1972.

-define- year as v1Ø and inflate as .Ø5 (constants can be -define-d as well). Set year ← 72, v(year) ← .25 and pos ← 1Ø1Ø.

Generate each line in the table by using the following calc.

```
calc    v(year+1) ← v(year)+v(year)xinflate
```

and -showt- year and v(year) and then add 1 to year and 1ØØ to pos.

In the above -calc- what variables are used? ____________________________.

Use the directives "s" (save) and "is" (insert save) to duplicate the code. Ask your proctor if you are not familiar with these directives, or read the directions by pressing -HELP- on the line display page.

Below is what the table should look like:

Table of Inflation Assuming 5% Rise Per Year

Year	Cost of Bread
72	$Ø.25
73	Ø.xx
74	Ø.xx
75	Ø.xx

j) pptest for PP4 is next.

Programming Problem 5: Student Branching

This programming problem is designed to show how authors can provide for branching which is initiated by the student. The following commands will be needed to solve the problem. Read about them in "tutor" and in Sherwood V 1-10.

term	data	base	help
next	end	back	lab

a) Up to now you have had to work through your lesson in a "linear" manner when in student mode; that is, before getting to PP4 you had to work through PP1, PP2, and PP3. The first objective of this programming problem is to provide a quick way to get to a particular unit in student mode. The TUTOR command -term- should be placed in the particular unit with an appropriate tag; in student mode one can then jump to that particular unit from any place in the lesson at any time if one knows the tag of the -term- command. (Historically, the -term- command was invented so that a student could request a definition of a "term" at any time during a lesson; -term- is now more typically used by authors to move quickly from one place to another within a lesson.)

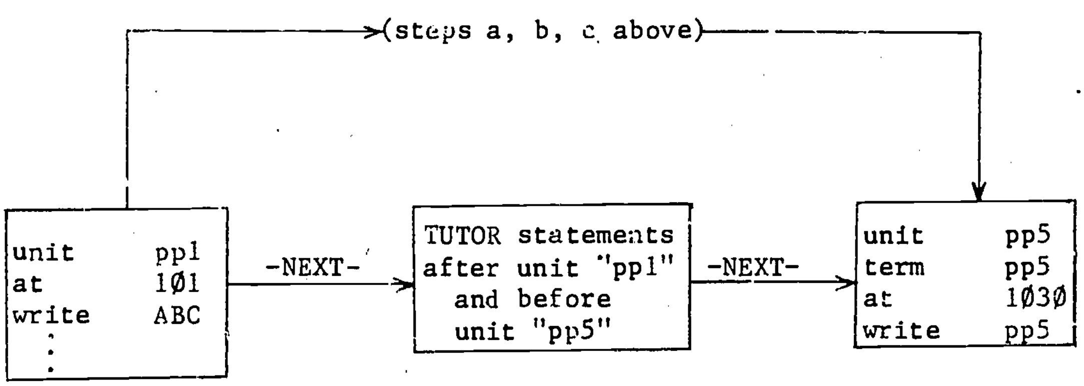
Place the following TUTOR statements after all of the TUTOR statements currently stored in your lesson space.

```
current TUTOR      unit     pp1
statements in        :
lesson               :

add these          unit     pp5
statements at      term     pp5
the end of your    at       163Ø
lesson space       write    pp5
```

Go into student mode. Now, from any display in your lesson, the following series of key presses will allow you to immediately branch to the TUTOR statements contained in unit "pp5".

a) press key: -TERM-
b) type: pp5
c) press key: -NEXT-



You should now be viewing the characters "pp5" at screen location 1030. Notice that you were able to by-pass all of the TUTOR statements between unit "pp1" and unit "pp5". The important thing to know is that the system "remembers" from where the -term- request was made.

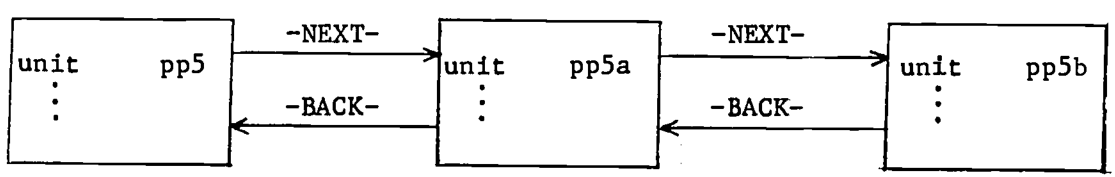
Press the -BACK- key

Notice that you have returned to the display generated by unit "pp1". Press -NEXT- and go to the display generated by unit "pp2" and "term" again into unit "pp5". Press -BACK- again and you will be returned to the unit from which the term request was made.

b) The -base- command gives the lesson author the option to alter the automatic return feature of any term- or help-type sequence. A -base- command with a "blank tag" causes the TUTOR system to clear the pointer used when the student presses the -BACK- or -BACK1- key. The author has instructed the system to forget from where the student initiated the help-type sequence.

Place a -base- command with a blank tag in unit "pp5". Enter student mode again and term (procedure outlined on the first page of this programming problem) to unit "pp5". Now press -BACK- and you will notice that you are <u>not</u> returned to the unit where you initially pressed the -TERM- key.

c) Construct two more units: "pp5a" and "pp5b". Include a -write- statement that gives the name of each unit when viewed during student mode execution. Using the -next- and -back- commands, provide TUTOR statements that allow an observer to -NEXT- and -BACK- between units "pp5", "pp5a", and "pp5b" as indicated below.



d) The keys labeled HELP, DATA, and LAB (and these keys shifted, called HELP1, DATA1, and LAB1) offer six possible student-initiated help-type branching sequences. The tag of each command specifies the name of the unit to jump to if the student presses the appropriate key. Pressing the key begins a "help" sequence which may be one or more units long. The last unit in the "help" sequence should contain an -end- statement so that the student is returned to the unit from which he requested the help (unless altered by a -base- command). For example, for the diagram below, unit "pp5" ashould contain the TUTOR statement

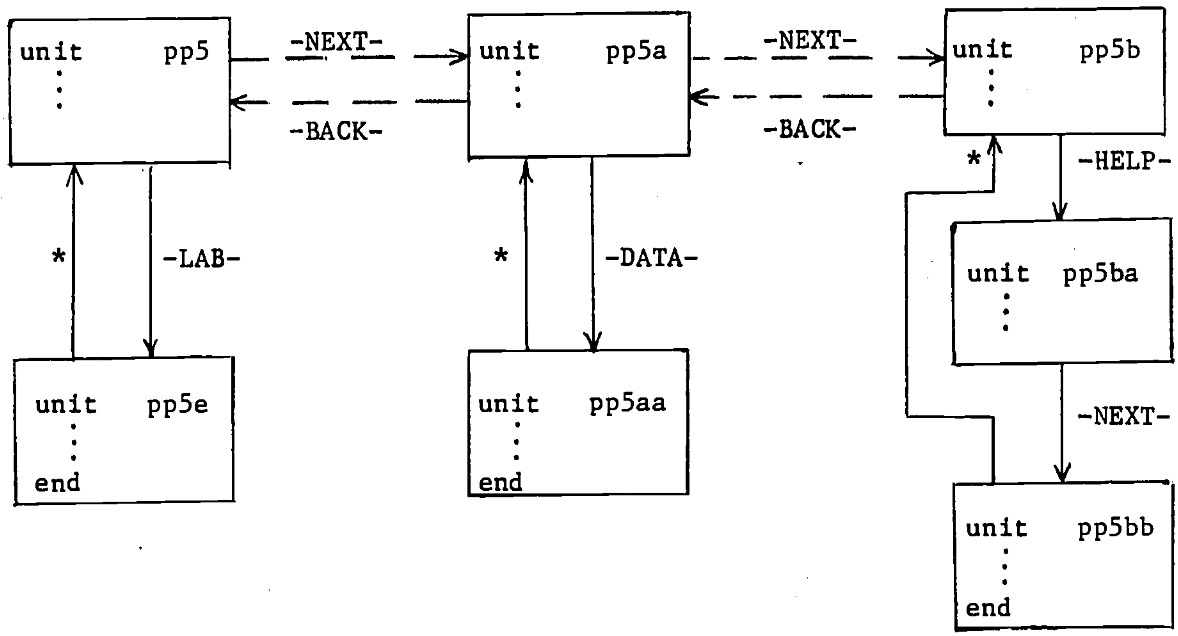
```
lab     pp5e
```

Unit pp5e should contain as the last command the TUTOR statement

```
end
```

Next you should add the necessary units and commands to your TUTOR code so that the structure of the lesson follows this diagram:

(Provide the -write- statements that identify each unit.)



* The units "pp5e", "pp5aa", "pp5ba", and "pp5bb" are all reached via keys that produce a help-type sequence. In help-type sequences, one may return to the "base" unit by pressing either the -BACK- (if a -back- command has not been encountered), -BACK1-, or -NEXT- (if an -end- command with a blank tag or an -end- command with a tag "help" has been encountered) keys.

e) Undoubtedly, as a PLATO author you are now aware that it is possible to become confused easily about the flow of ideas to the student, as illustrated by the "flow diagram" in part d. Of course, there are ways to cope with this problem of getting lost in the maze of possible sequences of units the student might follow. For example, in the flow diagram in part d, you may have noticed that the units had names that look like "family names": pp5, pp5e, pp5a, pp5aa, pp5b, pp5ba, pp5bb. Obviously, you as a PLATO author can interpret this as a hierarchy (pyramid) in pp5 → programming problem 5 which means the fifth of the programming problems. And, as in an outline, we would have

```
  I.  pp5
      a.  pp5e
 II.  pp5a
      a.  pp5aa
III.  pp5b
      a.  pp5ba
      b.  pp5bb
```

(Of course, presenting ideas on PLATO is more flexible and lively than in a written report, but the organizational idea is the same.)

The unit names for the units named pp1, pp2, pp3, were chosen simply to denote their order. Sometimes, though, you may want to name your units so that their names reflect the "contents" of the unit instead of (or in addition to) their "position" in the sequence of lesson materials. Suppose you are teaching about three animals: lions, tigers, and bears -- and that you have several units for each of them. You might incorporate an outline structure into the unit names as shown below.

```
  I.1.  lions1
        a.  lions1a
        b.  lions1b
    2.  lions2
    3.  lions3
        a.  lions3a
 II.1.  tigers1
    2.  tigers2
        a.  tigers2a
III.1.  bears1
    2.  bears2
        a.  bears2a
    3.  bears3
        a.  bears3a
        b.  bears3b
        c.  bears3c
```

Just think of yourself as Adam in the Garden of Eden and that <u>you</u> as an author must remember the names you give your units! You'll do all right.

Programming Problem 6: Author Branching

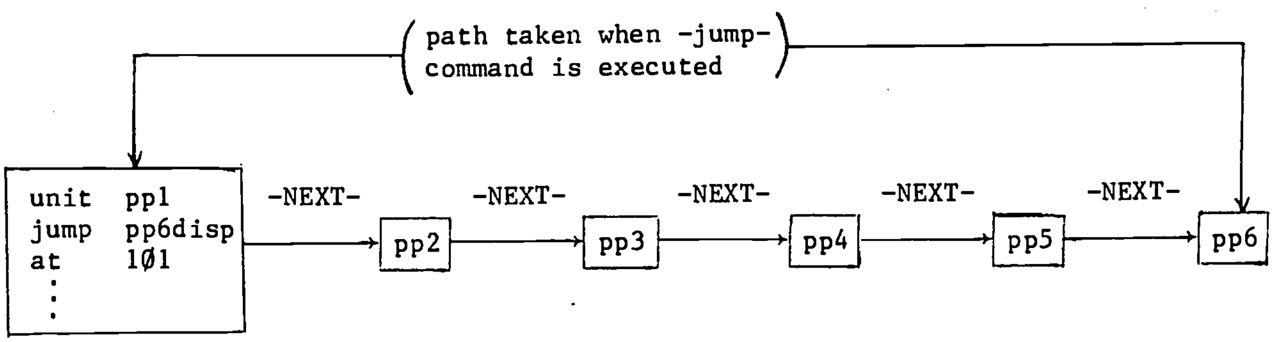
In PP5 you studied the commands which provide for branching via student control. PP6 will illustrate the use of several author branching commands. The following commands will help solve this problem.

unit	jump	write
at	do	pause

a) In PP5, part a, the -term- command was introduced as a method to skip a section of lesson code. The -jump- command provides another means for skipping sections of code. In the code for PP1, place the following -jump- statement.

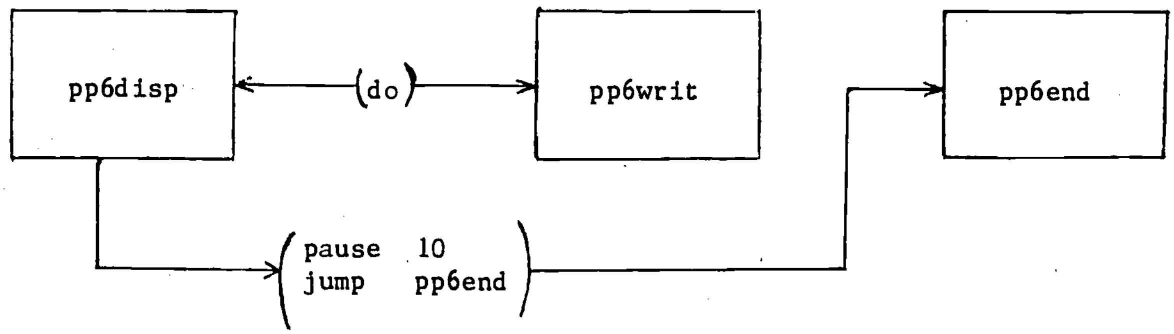
```
unit     pp1
jump     pp6disp    $$insert this line
at       1Ø1
  .
  .
  .
```

Now when you finish part b and enter your lesson via -STOP1-, you will be placed directly into the TUTOR code for PP6.



b) Create a -unit pp6disp- which, using -at- and -do pp6writ- statements, places the same complicated writing at 5 different screen positions. To do this, start a second -unit pp6writ- which contains the complicated -write- statement (not necessarily long, just subscripts, backspaces, etc. -- like $X_1^2 + 4X_1X_2 - 5X_2^3$ or SO_4^{-2}).

End the -unit pp6disp- with a -pause 10- command followed by a -jump pp6end- command. Physically locate the -unit pp6end- after -unit pp6writ-. Create an appropriate display for this unit. A flow chart of the above is as follows:


pp6disp
do
pp6writ
pp6end
pause 10
jump pp6end

Programming Problem 7: Conditional Sequences

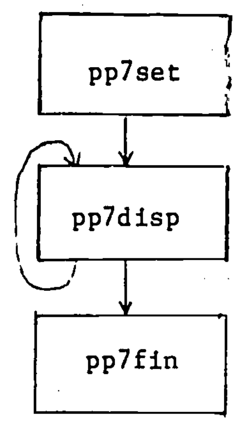
In PP7, conditional forms of TUTOR statements will be studied. The following commands will help solve this problem.

unit	calc	next	jump
at	writec	goto	

a) The purpose of this section is to construct a unit that -next-'s to itself four times. The first time the unit is viewed, a message should be displayed at screen position 1Ø15 that says, "First time for unit PP7". During succeeding times that the unit is viewed, the following messages should be displayed two lines lower than the preceding message (i.e., at 1215, 1415, etc.):

time in unit	screen message
first	First time for unit PP7
second	Second time for unit PP7
⋮	⋮
fifth	Fifth time for unit PP7

To do this start a new unit "pp7set" which initializes the variables needed for the -at-, -writec-, and -next- commands. At the end of pp7set, include a -goto- statement to unit "pp7disp".



In unit "pp7disp", use an -at- command whose tag is incremented during each passage through the unit. Also, use a -writec- command to display the different messages, and a conditional form of the -next- command to control movement each time the -NEXT- key is pressed. When the -NEXT- key is pressed while viewing the last message, the viewer should be taken to unit pp7fin. In unit pp7fin, -write- an appropriate message to the viewer.

Important: For _this_ problem only, the -next- command _must_ be the last statement in unit "pp7disp". The location of the -next- statement in this position os important for parts c and d.

1) How many messages are seen at any given moment? ____________

2) How many key presses were required of the viewer for him to see all the messages? ____________

b) Check your solution to a by looking at PP7 in lesson "ppsoln".

c) Replace the -next- command with a -jump- command. The tag of the -jump- command should be identical to that of the -next- command it replaces.

1) How many messages are seen at any given moment? ______________________

2) How many key presses are required? ____________________

3) What is the difference between part a and c? ______________________

____

____

____

d) Replace the -jump- command with a -goto- command. Again the tag of the command should not be changed.

1) How many messages are seen at any given moment? ______________________

2) How many key presses are required? ______________________

3) What is the difference between part c and d? ______________________

____

____

____

e) Take the pptest for PP7.

Programming Problem 8: Iteration

In PP8 the process of looping or of repeatedly executing the same section of TUTOR code will be examined.* The iterative -do- command will be used to set and display several TUTOR variables. The following commands will help solve this problem. Read about iterative -to- in "tutor".

unit	show	do	calc
at	write	next	

a) Copy either (but not both) of the following into a unit called "pp8".

```
do      pp8row,v1⇐1,64          (or)      do      pp8row,v1⇐16Ø1,1664
unit    pp8row                             unit    pp8row
at      v1+16ØØ                            at      v1
write   a                                  write   a
```

b) In unit "pp8" insert a second -do- statement that will display a column of b's starting at screen location 132. (To do this you will have to add another unit -- call it "pp8col".)

c) Provide a method for getting from unit "pp8" to a new unit "pp8num". In this unit do the following:

(1) Use a -do- statement to set the TUTOR variables 26-50 equal to the numbers 1-25, respectively (i.e., v26=1, v27=2, etc.). Also, after each variable has been set, use a -show- statement to show the value in that variable. Do not use an -at- statement in either unit. NOTE: To see what your display should look like, see lesson "ppsoln" in student mode.

(2) Since the single -show- command in part 1 does not present the data in an easily understandable format, replace that command with a set of commands that display the contents of v26-v50 in column form:

```
v26 = XXXX
v27 = XXXX
. . . . . .
```

PLATO screen as seen in student mode.

HINT: To display this line you must do the following:

(a) display a "v"
(b) show the "26"
(c) write an "=" (with spaces)
(d) show the contents of v26.

d) Now take the pptest for PP8.

---

*This is called iterating.

Programming Problem 9: Simple Judging

In programming problems 1 through 8 you studied (a) display techniques, (b) student initiated branching, (c) author branching, and (d) use of TUTOR variables. The next seven programming problems deal with techniques used in asking a question and judging simple student responses. In PP9, you will be given the exact code to type into your lesson.

Start a new -unit pp9arr- and copy the following:

```
Command     Tag

term        pp9arr   $$Check with your proctor if you don't know -term-.
at          215
write       Geography:  State of Illinois
arrow       1Ø15
at          51Ø
write       What city is the capital of Illinois?
answer      <city,is,the,The> Springfield (Illinois,Ill)
write       Let us now consider other questions about the
             State of Illinois.
no
write       Erase your answer and try again.
```

Try all of these possible answers.

Student responses	Computer's reply
a) Springfield, Illinois	________
b) The city of Springfield, Ill.	________
c) Springfield is the Illinois city	________
d) Chicago	________
e) The Illinois city is Springfield	________
f) Springfield, Ohio	________
g) Springfield	________
h) Illinois	________
i)* Springfeild, Ill	________
j)*Spingfield, Illunois	________
k)*Spingfild, Ilinois	________
l)*Fieldspring, Noisilli	________

*misspellings

Answer these questions by referring to the segment of code above.

Which words in the tag of the -answer- command are to be treated as synonymous -- that is, interchangeable?

____

Which word(s) in the tag of the -answer- command are optional -- That is, may be present in the student's answer but are not required?

____

Which word(s) must be present in the student's answer if it is to be judged ok?

____

How does PLATO indicate to the student that a word is misspelled?

____

How does PLATO indicate to the student that his word order is wrong?

____

Remember lessons "ppsoln", "pptest", and "ppcode" are part of the programming problems.

Programming Problem 10: Anticipating Variety in Student Responses – Alternative Forms of a Correct Answer

The following diagram is supposed to represent what your students will see on the screen. It is your task as an author to write the TUTOR statements which will display the text and judge the student answers as shown. The following commands will help solve PP10 and also PP11.

unit	arrow	answer	wrong
at	write	no	

Start a new -unit- and insert the appropriate code after the -unit- statement to place the following question and arrow on the screen. Then generate the TUTOR code to handle the student answers and computer responses which appear below.

Simulation of
PLATO screen

The sum of five and three is _ _ _ _ _.

>

Possible student answers and computer's response:

	Answer	Judgment
a.	eight	ok
b.	8	ok
c.	15 We are working on addition, not multiplication.	no
d.	fifteen We are working on addition, not multiplication.	no
e.	(any other answer) The correct answer is eight.	no

Now take the pptest for PP10.

Programming Problem 11: Supplying Feedback on Wrong Choices

Start -unit pp11- and add the appropriate code. We suggest using the directives "id" and "sd" to form the screen display. The following commands will help solve PP11.

unit	arrow	answer	write
at	no	wrong	

Simulation of PLATO screen

```
Select the correct answer
by typing its corresponding
letter.

The engine component which acts
as the pivot point for the rocker
arms is the

a)  pushrods
b)  rocker arm shaft
c)  valve lifters
d)  valve stem


⊳
```

The <u>five</u> possible responses should be handled as follows:

1) a no

The pushrod applies a force to the rocker arm causing the rocker arm to pivot.

2) b ok

3) c no

4) d no

The valve stem moves along with one end of the rocker arm.

5) (anything else) no

Please select a, b, c, or d for your answer.

Go on to the test for PP11 after reading the following comment.

A Comment on Appropriate Feedback Messages

Notice that some of the students' wrong choices in the above exercise will result in not only a "no", but also a <u>feedback</u> message. The ideas behind these feedback messages are to help the student distinguish between the wrong choice he/she made and the correct choice. Sometimes, this "help" may merely mean telling the student why he/she was wrong as in the "messages" for wrong choices 1a and 4d above. At other times, you as the author will have to anticipate that the student may not be able to give the correct answer after a number of tries at the arrow. This might happen, for example, in an arithmetical problem requiring the student to calculate an answer or in a question requiring the student to give a missing word in a sentence or to identify a particular concept by its definition. In such cases, the student may "hang-up" in the lesson and not be able to proceed unless <u>someone</u> tells him the answer. That <u>someone</u> should be you as the author. Give the student <u>appropriate</u> feedback so that he/she will <u>eventually</u> be able to arrive at an answer to any question in the lesson, or at least so that the student may go through all the arrows in the lesson. One other point should be mentioned about feedback -- messages which insult, ridicule, etc. the student do not help the student learn. NEVER INSULT the student!

Programming Problem 12: Phrases in Student Responses

In PP12 you will be given a verbal description of the situation to be programmed. Your problem will be to generate the code which will correctly handle the student answers.

The following commands will solve this problem:

unit	answer	wrong	arrow
at	write	no	

Start a new -unit pp12- and starting at the third line, tenth space write, "Higher yielding corn is produced by inventing different." Accept student responses at the fifth line, twentieth space. If the student types an incorrect response containing the word "homozygous," then write on the seventh line, tenth space, "It is not possible to improve the yield by crossing homozygous individuals." The correct answers are "heterozygous varieties," "heterozygous strains," heterozygous generations," "hybrid varieties," "hybrid strains," and "hybrid generations." Any other answer given by the student should be considered as being incorrect and the computer should reply "Erase your answer and try again." Only one -answer- should be used.

Programming Problem 13: Modifying the Judging Process, the -specs- Command

PP13 studies how the answer judging process can be modified by the use of the -specs- command. Only five tags of the -specs- will be used in this programming problem. When studying the -specs- command in reference lessons, notice all of the different judging options which can be changed using the different tags of the -specs- command. The following commands will help solve this problem.

unit	arrow	write
at	answer	specs

Start -unit pp13spc- and ask the student, "What are the names for the four years of high school?" (Put in a -next pp13spc- to allow you to try the unit several times).

Correct answer: freshman sophomore junior senior

a) Without a -specs- try several responses with the order changed, with misspellings, extra words, etc. (Use the -EDIT- key to save retyping the responses).

b) Add a -specs bumpshift-. Try capitalizing the responses. (e.g., Freshman, sophomore, juNior, SENIOR).

Change the -answer- command to -answer freshman Sophomore junior senior-. What happens?

c) Replace the -answer- command of part b with the -answer- command of part a. Replace the previous -specs- with -specs noorder-. Try the responses in different order. (e.g., senior, freshman, junior, sophomore).

d) Replace the previous -specs- with -specs okextra-. Try responding with extra words. (e.g., the years are freshman, sophomore, junior and senior).

e) Replace the previous -specs- with -specs okspell-. Make spelling errors in the responses. (e.g., froshman, sophmore, juneor, Senior).

f) Replace the previous -specs- with -specs nookno-. Watch the reply that is made after judging the student's response.

g) The above five tags of -specs- can be used together for any combination of effects which are desired. Replace the previous -specs- with one -specs- command which will display an OK after the following response.

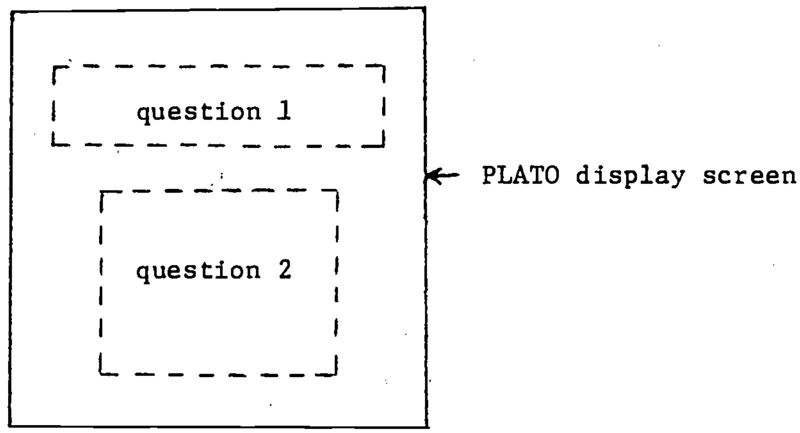
Senior, junior, sophmore, and freshman.

(Note: If there is more than one -specs- command for an -arrow-, only the tags of the last encountered -specs- command is in effect. However, a single -specs- command can have several tags and all of them will remain in effect together until the next -specs- command is encountered).

h) Take the pptest for PP13.

Programming Problem 14: Multiple Questions in One Display

This problem will require a student to answer two different questions on the same frame (display). The first question will be displayed for the student. When the first question is answered correctly, the second question will immediately be displayed for the student to answer. The general layout of the display screen is as follows:



Your task is to generate the code for the following displays. The following commands will solve this problem.

unit	arrow	endarrow	answer
at	write	wrong	no

Start a new -unit pp14- and add the appropriate code.

Part 1

Simulation of
PLATO screen

There are two major marking commands in the TUTOR language. Name these two commands.

>

Possible student answers and computer's reply:

	Answer	Judgment
a)	unit and arrow	ok
b)	arrow and unit	ok
c)	unit, arrow	ok
d)	arrow, unit	ok
e)	(Any other student answer) The TUTOR commands -unit- and -arrow- are the major marking commands.	

Part II

After the student answers question one, display the following information and question in the area designated for question 2.

Note: Use the -endarrow- command to separate the first question (and its answer-judging commands) from the second arrow (and all commands related to the second question).

(Area for first question)
unit and arrow

Type the letter associated with
the minor marking TUTOR command.

a.	write	d.	answer
b.	no	e.	wrong
c.	specs	f.	draw

Possible student answers for part II and computer's reply:

	Answer	Judgment
a)	c	ok
b)	a,b,d,e, or f Not the proper command	no
c)	(any other answer) Only a single letter a through f is a proper response.	no

Programming Problem 15: Conditional Feedback

In PP15, you will again be given a verbal description of the problem to be programmed. Your problem again will be to generate the code to properly handle the specified student answers. The following commands will help solve this problem.

unit	writec	wrong	endarrow	at
write	answer	calc	no	arrow

a) Start a new -unit pp15- and ask the student the following question: "The U.S. Great Lakes system contains ________ lakes (how many?)." Accept a student's answer two lines below. Accept student answers of "five" or "5".

b) When a correct answer has been given, ask the following question: "Of the five lakes, what is the name of the largest?" Accept either "Superior" or "Lake Superior" as correct answers. When the first incorrect answer is given, remind the student that the largest lake is the one that lies between the Michigan peninsula and Ontario, Canada. If the student answers incorrectly a second time, tell him that the correct answer is "Lake Superior."

Hint: For the -writec-, the Universal separator is "micro" ",".

Example: writec n10↕minus↕zero↕positive

Also: Don't forget to use the -endarrow- command.

c) Take the pptest for PP15.

This is the last programming problem. You may wish to review PP1-14 and/or the pptests before taking the final exam.

Appendix II

Examples of the Programming Problem Tests
to accompany the
MTC Author Training Course

The following pages reproduce displays of some of the on-line tests administered to new authors. The number of the programming problem test (PPT) corresponds to the number on a programming problem. The part of the display with a gray background is computer-generated and hence different for each student. In most of the examples shown, author responses have been simulated in order to show the type of feedback given. All of the tests are available on-line in lesson "pptest".

c) Given the following TUTOR -at- and -write- statements, how many screen lines (as seen by the student) will be used during execution in student mode?

i) at 222Ø display 4Ø characters and spaces. How many screen lines will the student see?

1 ok

ii) at 224Ø display 4Ø characters and spaces. How many screen lines will the student see?

2 ok

iii) at 226Ø display 4Ø characters and spaces. How many screen lines will the student see?

3 no

feedback → 12345
678900
12345
67890
12345
67890
12345
67890

The following set of -calc- statements are in a unit:

1. calc v3 ⇐ 5
2. v1 ⇐ v6 ⇐ 11
3. v2 ⇐ v6 + 2 x v1
4. v (v3) ⇐ v1
5. v1∅ ⇐ v3 + 2^2
6. v (v1 + 1∅) ⇐ v1 + v1∅

Of the variables v1 to v15∅, what is the highest numbered variable used?

v21 ok

Of the variables v1 to v15∅, what variable holds the largest value?

v2 ok

What is that value? 3∅ no

feedback after several errors → The correct answer is 33.

HELP available.

Look at the -calc- sequence again:

1. calc → v3 ⇐ 5
2. → v1 ⇐ v6 ⇐ 11
3. → v2 ⇐ v6 + 2 x v1
4. → v (v3) ⇐ v1
5. → v10 ⇐ v3 + 2^2
6. → v (v1 + 10) ⇐ v1 + v10

First v3 is set to 5.

Second, v1 and v6 are set to 11.

Third, v2 is set to v6 + 2 x v1 = 11 + 2 x 11 = 33.

Fourth, v (v3) = v (5) = v5 is set to 11.

Fifth, v10 is set to v3 + 2^2 = 5 + 4 = 9.

Finally, v (v1 + 10) = v (11 + 10) = v21 is equal to v1 + v10 = 11 + 9 = 20.

The student pressing the -HELP- key on the previous page is shown this display. One line at a time is pointed to by the arrow and explained in a line of the text.

This is a test of the -showt-, -calc-, and -define- commands.

The following commands are part of a unit:

```
calc     v7 ⇐10.5
at       1718
showt    v7,5.2
```

Enter in each character-space box the character which will be displayed (include blanks):

1718
↓
| | |1|Ø|.|5| | | no
↑
↑

feedback → Character #8 is wrong.

Press HELP to see correct answer.

If the student presses -HELP-, he sees:

The following commands are part of a unit:

```
calc    v7 ⇐1Ø.5
at      1718
showt   v7,5.2
```

This is what the filled-in boxes should look like:

1718   1718
↓

			1	Ø	.	5	Ø		

There are 5 characters to the left of the decimal point. The first 3 characters are blanks.

There are 2 characters to the right of the decimal point.

Assume the following TUTOR code is being executed in student mode.

```
unit    alpha
end
*
unit    beta
help    alpha
data    gamma
*
unit    gamma
back    alpha
term    G
lab     delta
*
unit    delta
base
help    alpha
```

Assume the student begins in unit alpha and presses the following key(s). What unit is the student in (what is his main unit) when he has pressed:

Problem 1 (of 15)

-NEXT-

> alpha no

feedback → -end- only has an effect when in a HELP-type sequence

Answer with the word "hint" to get a hint.

Assume the following TUTOR code is being executed in student mode.

```
unit     alpha  ← ←always start here
end
*
unit     beta
help     alpha
data     gamma
*
unit     gamma
back     alpha
term     G
lab      delta
*
unit     delta
base
help     alpha
```

Assume the student begins in unit alpha and presses the following key(s). What unit is the student in (what is his main unit) when he has pressed:

Problem 4 (of 15)

-NEXT-, -HELP-, -BACK-

> hint ..ne

feedback → pressing -BACK- while in a HELP-type sequence when no -back- has been specified returns one to the base unit.

Answer with the word "hint" to get a hint.

The following is a set of TUTOR code:

```
1.  unit    one
2.  write   How are you today?
3.  unit    two
4.  at      4ø5
5.  write   Hi there!
6.  at      5ø5
```

<u>After</u> which line should the statement

```
    do      two
```

be inserted so that the student display in unit one will be:

4ø5
↓

Hi there!
How are you today?

> 4 no

feedback → An execution error will result if this line of code is placed anywhere in unit two because of a fatal -do- loop (a unit cannot -do- to itself infinitely).

Now look at the following code:

```
 1.  unit   d
 2.  write  d
 3.  do     e
 4.  do     d
 5.  unit   z
 6.  write  z
 7.  unit   g
 8.  write  g
 9.  jump   z
10.  unit   e
11.  write  e
12.  do     p
13.  jump   z
```

Execution starts at line 1.

In which unit will the student be when the computer stops executing this section of code?

> g   no

feedback→ Wrong answer again.
Press -DATA- to see the answer by
executing the code.

You have finished test T-PP6.

You made 20 attempts on the 5 questions.

You should practice using -do- and -jump- commands more.

Consider the commands -next-, -goto-, -do-, and -jump- when answering the following questions.

What command(s) could -xxxx- logically be?

A branch executed via the -xxxx- command(s) will erase the screen.

> jump next do no

feedback → neither -do- nor -goto- erase the screen

Consider the commands -next-, -goto-, -do-, and -jump- when answering the following questions.

What command(s) could -xxxx- logically be?

```
unit    unitone
write   abc
xxxx    unittwo
write   def
unit    unittwo
```

> jump next   no

feedback → With -jump- the -write def- would never get done.

Create the proper -do- statement to make a vertical column of b's at the right-hand edge of the screen as shown. The variable v1 will be used for the -at- value.

```
unit    bee
do      loop, v1 ⇐164, 3264  no
unit    loop
at      v1
write   b
```

Make the first "b" appear at 164.

feedback
The second time through the loop v1 will be 165 which is an illegal value for the -at- command.

```
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
```

Create the proper -do- statement to make a vertical column of b's at the right-hand edge of the screen as shown. The variable v1 will be used for the -at- value.

b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b

```
unit   bee
do     loop,v1 ⇐164,3264,-100   no
              Δ     ⇐
unit   loop   ↑     ↑
at     v1   feedback is response mark-up
write  b
```

Now make the b's be displayed from the bottom up, rather than from the top down. (like this, press NEXT).

The feedback symb[illegible] [illegible]ndicate something is missing before the "164" and that "3264" is out of order and should appear farther to the left. The student should have placed "3264" before "164."

Create the proper -do- statement to make a vertical column of b's at the right-hand edge of the screen as shown. The variable v1 will be used for the -at- value.

```
  unit     bee
> do       loop,v1 ⇐164,3264,200    ok
  unit     loop
  at       v1
  write    b
           b
```

Now suppose we changed the -write- command so that it writes <u>two</u> lines of b's at once (as above). Again create a -do- statement which will write the vertical string of b's.

feedback
Your answer is all right, but beware: You should have 3164 as the final location to stop the -do- loop. In this case having 3264 instead of 3164 doesn't hurt, but this type of mistake can sometimes cause problems.

```
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
b
```

Now fill in the tag of the -at- command to make the display shown in the upper right-hand corner of the screen.

v
e
r
t
i
c
a
l

```
unit     vertical
do       looper,v1⇐Ø,7
unit     looper
at       v1 x 164   no
writec   v1,,v,e,r,t,i,c,a,l,
```

feedback

With that -at- command, you would get these locations:

Ø, 164, 328, 492, 656, 82Ø, 984, 1148.

You should get 164, 264, 364, 464, 564, 664, 764, 864.

Now fill in the tag of the -at- command to make the display shown in the upper right-hand corner of the screen.

v
e
r
t
i
c
a
l

```
unit     vertical
do       looper,v1 ⇐Ø,7
unit     looper
at     > v1 x 1ØØ + 64    no  ←  Any mathematical expression
                                  which equals 100vl + 164 is
writec   v1,,v,e,r,t,i,c,a,l,     counted correct.
```

feedback
With that -at- command, you would get these locations:

64, 164, 264, 364, 464, 564, 664, 764.

You should get 164, 264, 364, 464, 564, 664, 764, 864.

During this test, you will be given one question at a time to a total of 9 questions. All the questions refer to the code below. If the answer to a question could be any of a great number of possibilities, then enter "can't tell." After 3 incorrect answers, the correct answer is displayed.

```
 1.  unit    tpp1Ø
 2.  at      51Ø
 3.  write   interesting facts
 4.  arrow   1215
 5.  at      1Ø15
 6.  write   What giza mifraz was it?
 7.  answer  (tif,mif,wif)
 8.  at      1415
 9.  write   zat so mort
10.  wrong   fram
11.  at      1415
12.  write   got zotz brif?
13.  answer  tollyus
14.  no
15.  at      1415
16.  write   gamit dry aten
```

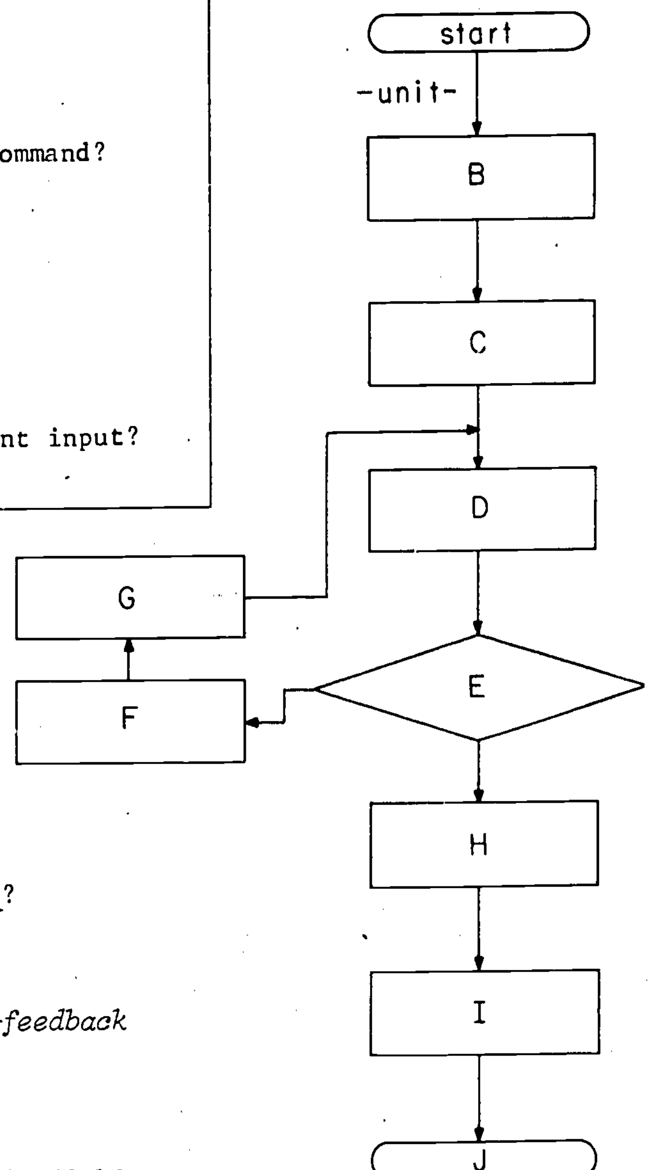
Problem 2 (of 9)

student entered	judgment was	reply was
mif		> got zotz brif? no

feedback → line 7 is matched

TUTOR STATEMENTS IN QUESTION

```
 1.  unit     pp11
 2.  at       528
 3.  write    TUTOR
 4.  arrow    1324
 5.  at       816
 6.  write    -write- is what type of command?
 7.  answer   regular
 8.  at       1518
 9.  write    Great!
10.  wrong    judging
11.  at       1518
12.  write    Does a -write- need student input?
                        Try again.
```



PPT11 will test the order of execution of the above code and when things appear on the screen. Press -DATA- at any time to run the above code to answer any question. Watch the flowchart on the side. This will change as you answer the questions.

At which command does the computer start?

unit ok

-unit- is a major marker in TUTOR and is shown on the flowchart to remind you it is a marker.

←*feedback*

-DATA- to run above TUTOR code. -HELP- Available.

The test on PP16 consists of 19 questions: 9 in the format below; 10 are multiple choice.
NOTE: You get only one chance at each question.

In this part, the correct answers can be found among the following:
nookno, bumpshift, okextra, noorder, okspell.

GIVEN:

a) the commands... specs [?]
answer congnitive attitudinal psychomotor

and

b) the student response → attitudinal, cognitive, psychomotor

QUESTION:
What single -specs- tag is required to get an "ok" judgment?

2 ⋗ okextra no

feedback → "noorder" -- NO unique word ORDER is required.

The test on PP13 consists of 19 questions: 9 in the format below; 10 are multiple choice.
NOTE: You get only one chance at each question.

In this part, the correct answers can be found among the following:
nookno, bumpshift, okextra, noorder, okspell.

GIVEN:

a) the commands... specs [?]
answer multiple choice

and

b) the student response → Multiple Choice

QUESTION:
What single -specs- tag is required to get an "ok" judgment?

1 ≫ noorder no

feedback→ "bumpshift" -- BUMP any SHIFT codes.
Capitalization in the response is ignored.

Rearrange the lines of code on the right so that unit "math" will execute in the manner you saw. The first five lines of the unit are on the left. Type only the number of the line you want.

```
unit     math
calc     n1∅ ⇐ -1
at       1∅1∅
write    2 + 3 = ?
arrow    1∅16
-----------------
answer   5
wrong    6
```

>

The student types the number of the lines he wants next; the computer moves it over to the left and crosses it out on the right.

Randomized

```
 1.  no
 2.  answer    12
 3.  at        2∅2∅
     write     3 x 4 = ?
 4.  calc      n1∅ ⇐ n1∅ + 1
 5.  writec    n1∅,Try again.,
               Answer is 5.
 6.  write     Don't add!
---7.---wrong---6----------------
 8.  arrow     2∅25
 9.  endarrow
10.  wrong     7
11.  write     Don't multiply!
--12.---answer--5----------------
```

You may change your answers later if you wish.
Or press -DATA- if you want to start over.

Rearrange the lines of code on the right so that unit "math" will execute in the manner you saw. The first five lines of the unit are on the left.

```
     unit     math
     calc     n1Ø ⇐ -1
     at       1Ø1Ø
     write    2 + 3 = ?
     arrow    1Ø16
-----------------------------
 1.  answer   5
 2.  wrong    6
 3.  write    Don't multiply!
 4.  no
 5.  calc     n1Ø ⇐ n1Ø + 1
 6.  writec   n1Ø,Try again.,
              Answer is 5.
 7.  at       2Ø2Ø
     write    3 x 4 = ?
 8.  arrow    2Ø25
 9.  answer   12
10.  wrong    7
11.  write    Don't add!
12.  endarrow
```

Randomized

~~---1.--write---Don't-multiply!----~~
~~---2.--arrow---2Ø25---------------~~
~~---3.--wrong---6------------------~~
~~---4.--no-------------------------~~
~~---5.--answer--5------------------~~
~~---6.--answer--12-----------------~~
~~---7.--wrong---7------------------~~
~~---8.--calc----n1Ø⇐-n1Ø-+-1------~~
~~---9.--at------2Ø2Ø---------------~~
~~---write---3-x-4-=-?----------~~
~~--10.--writec--n1Ø,Try-again.,----~~
~~-----------Answer-is-5.--------~~
~~--11.--write---Don't-add!---------~~
~~--12.--endarrow-------------------~~

There are twelve different combinations of these twelve commands which will produce correct results. Feedback to the author is based on the the ordering he seems to be trying to achieve.

Line 5 is the first incorrect line.
Press -NEXT- to try again, or:

-DATA- to run the unit correctly.
-LAB- to run the unit as you have assembled it.

The code as you have arranged it is incorrectly ordered. Enter the line numbers of the code in the correct order. The first five lines of the unit are on the left.

```
unit      math
calc      n1Ø ⇐ -1
at        1Ø1Ø
write     2 + 3 = ?
arrow     1Ø16
------------------------------
answer    5
wrong     6
write     Don't multiply!
no
writec    n1Ø,Try again.,
          Answer is 5.
calc      n1Ø ⇐ n1Ø + 1
endarrow
at        2Ø2Ø
write     3 x 4 = ?
>
```

Your Incorrect Order

~~---1.--answer------5-----------------~~
~~---2.--wrong-------6------------------~~
~~---3.--write-------Don't-multiply!----~~
~~---4.--no-----------------------------~~
~~---5.--calc--------n1Ø-⇐-n1Ø-+-1------~~
~~---6.--writec------n1Ø,Try-again.,----~~
~~--------------Answer-is-5.--------~~
~~---7.--at--2Ø2Ø-----------------------~~
~~--write-------3-x-4-=-?----------~~

8. arrow 2Ø25
9. answer 12
10. wrong 7
11. write Don't add!

~~--12.--endarrow-----------------------~~

You may change your answers later if you wish.
Or press -DATA- if you want to start over.

Appendix III

Lesson "tutor"

The following pages reproduce some of the displays seen by authors using the on-line reference lesson, "tutor." To demonstrate its interactive characteristics, author responses have been simulated.

Type the name of the command for which you wish to see an explanation.

The commands that can be found in this lesson are shown now; listed both alphabetically and functionally.

>

Press -DATA- for "topics"
(See AIDS on all commands for further information.)

ALPHABETIC

answer	goto	put
ansv	help	rotate
arrow	ignore	showt
at	inhibit	size
back	jump	specs
bump	lab	store
calc	long	subl
circle	match	term
data	mode	unit
define	next	write
do	no	writec
draw	ok	wrong
end	pause	wrongv
erase		zero

REGULAR					JUDGING		
GENERAL	DISPLAY	CALC'S	AUTHOR BRANCH	STUDENT BRANCH	I	II	III
unit	at	calc			bump	answer	ok
*	write	addl	jump	next	put	wrong	no
arrow	writec	subl	goto	back	specs	ansv	ignore
long	showt	zero	do	help		wrongv	match
	draw	define		data		store	
	circle			lab			
	mode			term			
	erase			end			
	inhibit						
	size						
	rotate						
	pause						

The main display page of "tutor"

Press	for
a	Terminology
b	Judging
c	Variables
d	Expressions
e	Grid System
f	Conditional Form
-BACK-	index

Pressing -DATA- from the main display page gives a list of discussions which apply to many commands.

TUTOR Student Variables

Each student has 15Ø storage locations (called TUTOR student variables) available for individualization of his lesson material. These variables are "named" v1, v2, v3,...v15Ø (v stands for "variable").

Student variables may be used to store numbers (like 5, -3.124, or 51739234) or groups of characters (like "John", or "May 1917").

Any command which uses a numerical value in its tag (such as -at-) may also use a student variable in the same place. Since the value of the student variable can be changed during the lesson (by -calc-, -add1-, -sub1-, -zero-, -store-, or -match- commands), these commands are made much more powerful.

The value of a variable may be shown on the student's display by the -showt- command. The -writec- command and most of the branching commands can use the values of student variables to present material in a unique form or order to each student.

Press -NEXT- for expressions, -BACK- to return
-DATA- for a detailed explanation of variables
(for those people who wish they didn't have to
study variables, hated math, etc., try -DATA-)

Choosing item "c" on the previous display (p. III-2) gives this explanation. If the new author presses -DATA- at this display, he will go to a separate lesson which discusses TUTOR variables in detail.

ansv , wrongv

(judging)

PRE-REQS "judging" (press -DATA-)

EFFECT Current judge copy is compared to statement's tag. If the student's answer is within the specified tolerance, an *ok* judgment is made for an -ansv- command and a *no* judgment is made for a -wrongv- command.

TAG The tag has two parts. The first part specifies the correct answer and the second indicates the acceptable ± range. The following conventions are used when specifying the range value:

range tag:	acceptable answers:
a) no tag	tolerance of Ø, exact match.
b) number	author answer (AA) ± number
c) (number)%	AA ± (AA x n%)

EXAMPLES

```
unit    cor-ans
:
arrow   2845
ansv    5Ø
ansv    5Ø,1
write   close enough
wrongv  5Ø,5%
write   close!
```

Try the example at the left, the correct answer is the number 5Ø.

> 52 no ← *student selected value*

feedback → close!

The new author may try various values to test the judging characteristics of these commands.

bump

(judging)

PRE-REQS "judging" (press -DATA-)

EFFECT removes characters from the student's response prior to attempting a match

TAG a list of characters (1Ø characters max.)

EXAMPLES

```
bump    ) -    $$ removes all right parens, spaces, and
      ↑           dashes (minus sign) before attempting
    space         a match
```

by using bump) - all of these responses --

→ abc → a) b) c) → a-b-c → a)-b)-c)

become: abc

NOTES 1) to bump capitalization, se -specs bumpshift- or capitalize one of the characters in the tag, for example,

bump)Abc $$ removes: right parens, caps, a, b, c

Continued

Enter a sample student response and then the tag of the bump command. The result is what would be available for an attempted match after the -bump- "operated" on the sample response.

sample response (10 char's max for this lesson) ab$_c$dEf ok

-bump- tag (10 char's max always) ≫ cA

student supplied input

result → b$_d$ef

The new author may experiment with various tags and responses in order to understand some subtleties of this deceptively simple command.

writec

PRE-REQS "conditional forms" (press -DATA-), -at-, -write-

EFFECT writes one (1) of several phrases depending upon the value of the expression

TAG a variable, a comma, then one or more phrases separated by commas (or other delimiters)

EXAMPLES

```
at      51Ø
writec  v1,to,too,,two, ,
```

value of variable v1	what is displayed
negative	to
Ø	too
1	nothing (no spaces either)
2	two
3 or greater	one space

NOTES

1) `writec  v1,abc,    $$ abc  always written`
   `writec  v1,abc,,   $$ abc  if n1=neg,else nothing`
2) the first character after the expression identifies the delimiter to be used. Delimiters may be commas (,) or semicolons (;), one <u>OR</u> the other. See AIDS for the writec delimiters (♦).
3) multiply line messages may be written if desired by omitting the delimiter at the end of a line.

Continued

Given the following statements:

```
at      2ØØ5
write   The
writec  v1,,cow was, cows were,
writec  v2,, walking, running,
write    to the
writec  v3,, barn., water because it was
        such a hot day.
```

Enter values for v1, v2, v3 and note the effect.

```
v1 = ≫ 1   ok
v2 = ≫ 2   ok   student selected values
v3 = ≫ 3   ok
```

feedback
The cows were running to the water because it was
such a hot day.

-LAB- to exit.

By selecting values for the variables listed, the author controls the sentence to be written.

Given the following statements:

```
      at      2ø05
      write   The
      writec  v1,, cow was, cows were,
      writec  v2,, walking, running,
      write    to the
      writec  v3,, barn., water because it was
              such a hot day.
```

enter values for v1, v2, v3 and note the effect.

```
v1 =    1    ok
v2 =    ø    ok — student selected values
v3 =   -1    ok
```

feedback
The cows were walking to the

-LAB- to exit.

match

PRE-REQS [-arrow-, -writec-] for example

EFFECT attempts to find in the student's response the first occurrence of any one of a given list of words

TAG a variable in which to store the result of the attempted match, a comma, a list of words (separated by commas) which may occur in the student response.

EXAMPLE

```
arrow   3112
match   v14,cat,dog,Mouse,house
writec  v14,???,cat,dog,Mouse,house
```

NOTES In the example above v1 is set to -1 (and "???" will be written) only if the student's answer contains none of the words "cat", "dog",...etc. If the student response contains the word "cat" anywhere in it, v14 is set to Ø and "cat" is written. A response containing "dog" sets v14 to 1. Below is an arrow which executes the example above. Try these examples plus any of your own: "dog", "dog house", "house dog", "doghouse", "doll house".

The new author may experiment by entering different answers to see which word the computer finds first.

lab data help
lab1 data1 help1

PRE-REQS: -end-

EFFECT: When a student who is in a unit containing a -lab- command presses -LAB-, he is immediately "jumped" to another unit. Ditto for -help-, -data-, -lab1-, etc.

TAG: name of the unit to branch to

EXAMPLES:

```
data    dataunit

data    moredata
```

NOTES: -LAB1- means the -SHIFTED- -LAB- key, -HELP1- and -DATA1- are similar.

Although the branch that occurs when one of the above keys is pressed looks similar to a -jump-, there is an important difference: the student may return to the unit from which he began branching (called the base unit) in three ways:

1) he presses -NEXT- in a unit which contains an -end- command, but no -next- command.
2) he presses -BACK- in a unit which contains no -back- command.
3) he presses -BACK1- in a unit with no -back1- command.

COMMENT: To change the base unit, see -base- in "aids." These commands are often used in conditional form.

(Press -DATA- to see how)

Continued

This moves as the base unit changes base

```
unit   a
write  a
lab    c
unit   b
write  b
lab    d
next   d
unit   c
write  c
lab    e
unit   d
write  d
end
unit   e
write  e
next   b
```

main *This moves as the main unit changes*

DIRECTIONS

type r to restart in unit a
type e to read explanation again
type l to look up another command

SAMPLE EXERCISES

SCREEN → c

Beginning in unit a each time, press:

1) -NEXT- 4 or more times
2) -LAB- in unit b then several -NEXT-s
3) -LAB- in unit a, then several -NEXT-s
4) -LAB- in a, -LAB- in c, then -NEXT-s
5) -LAB- in b, then -BACK1-
6) -LAB- in a, then -NEXT-, then -BACK1-
7) -LAB- in a, -LAB- in c, then -BACK1-

In the above example, because there is no -back- command, -BACK- operates like -BACK1-

Normally, the process of student branching involves several "invisible" changes. This simulation demonstrates the computer's actions.

long

PRE-REQS "judging" (press -DATA-)

EFFECT prevents additional student input after a specified number of characters have been typed

TAG a number specifying how many characters may be typed

EXAMPLE long 3

type several keys (watch the screen)

> hou *student input*

NOTES 1) Since -long- is a "regular" command it must be before any judge-type commands after the arrow.
2) capitalized characters count as two characters, that is, a shift ( ) char. and the char. itself

COMMENT If no -long- is used -long 15Ø- is assumed. A -long 1- is treated specially: judging is forced automatically, also a shifted letter (really two characters) can be entered. For forcing judgment of other -long-s, see -force- in "aids".

The effects of a rather simple command may be seen here.

Appendix IV

Excerpts from "Documented Matching Drill"

Eileen Sweeney

January 23, 1974

What follows are pages from a "documented driver." Shown are the first page of general directions for use, the first page of programming (showing two units), a flowchart, and the line-by-line documentation for the first two units.

The following paper is a detailed documentation of a matching routine. The coding is divided into two parts, each part linked together by a driver unit consisting of a series of -join- commands. In the first section, the student matches two columns of concepts. In the second section, the student's responses are evaluated and various information is fed back.

The parameters used throughout the routine are set in unit define. Each author must supply both the number of matches the student is expected to make and the number of choices the student is given. The remaining parameters are optional and are turned "on" or "off" by setting them equal to 1 or Ø respectively. The first option *(ansall)* enables the author to force the student (by setting *ansall* equal to 1) to fill in all the matches before the sequence is evaluated. If *ansall* equals Ø ("off"), the student's responses may be evaluated at any time. The author must also set the evaluation options to Ø or 1. These options are *shwper* (the percentage correct), *shwrong* (show the student's incorrect responses), *shwrght* (show the correct response for each incorrect student response). Any or all of these options may be turned on or off. The author must also supply the correct series of responses in the tag of the -pack- command. Finally, the author should fill in the columns of concepts the student must match, and state the relationship between the two columns.

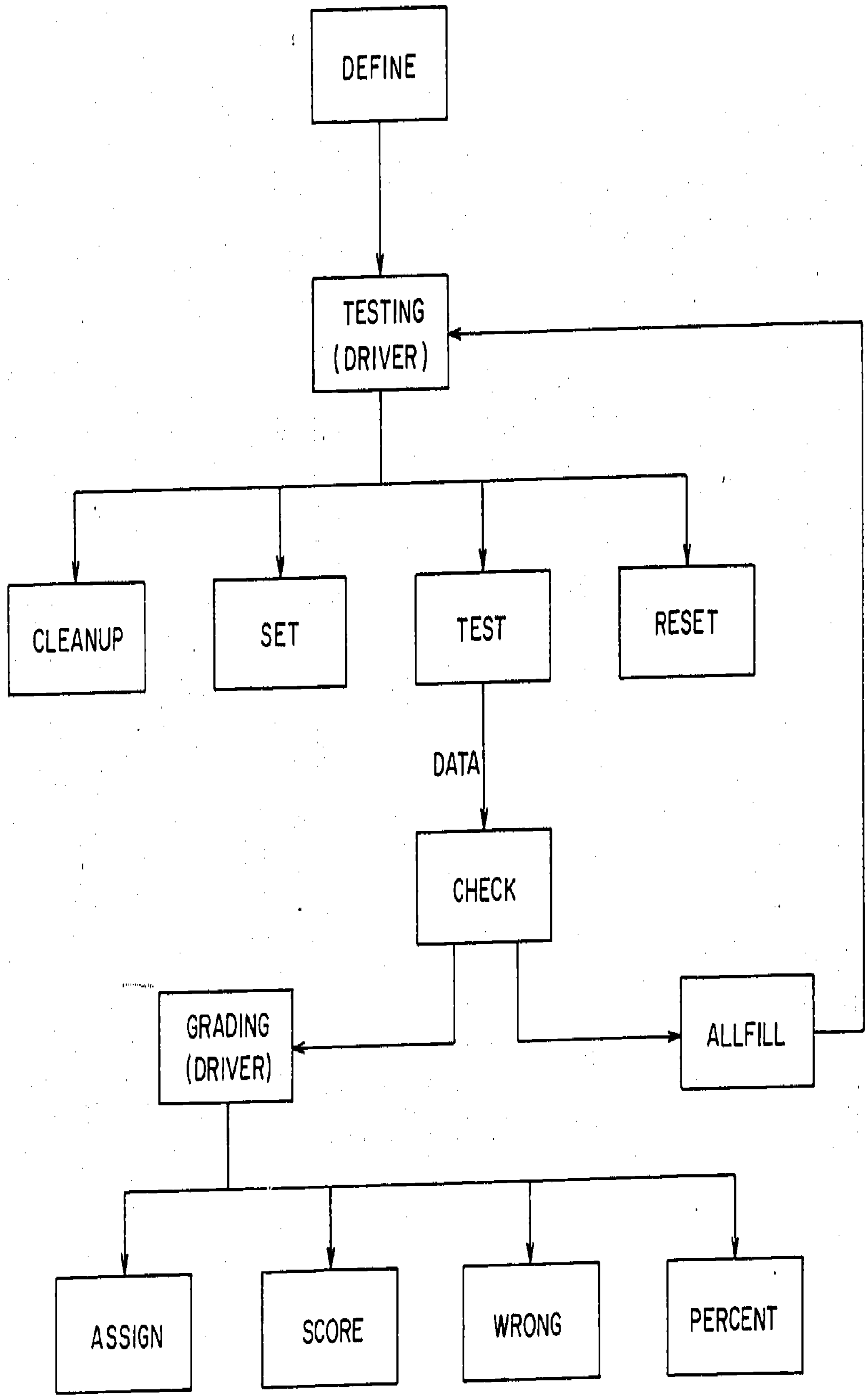
A number of instructional guidelines should be considered when writing the matching lists. State clearly the relationship between the two lists (i.e., tool to machine, part to function, etc.) Be sure and show only one relationship in any one matching drill. For example,

strong	skunk
	Hercules
	whiskey

presents a variety of types of strength (physical strength, odor, taste). Remember, too, that the relationship implies a consistent grammatical structure within each list. Use more "answers" than "questions", and, wherever possible, use the same answer twice (remembering to tell the student answers may be used more than once). In these ways the drills will be more difficult and the grading more reliable. Students must weigh each question separately without using a "process of elimination." Finally, avoid "question" lists of more than 10 or 12 items, or more than 16 "answers". Longer matching drills are better broken into smaller pieces with a "breather" in between.

```
unit     define
inhibit  erase
zero     n1,14
define   tempans=n1      $$temporary storage for student answer
         quesno=n2       $$question number
         matches=n3      $$number of matches student will make
         choices=n4      $$number of choices student has for matc
         arowloc=n5      $$screen location of judging arrow
         score=n6        $$total number of correct responses
         ansall=n7       $$(opt.)all questions must be answered
         swhper=n8       $$(opt.)show percentage correct answers
         shwrong=n9      $$(opt.)show student's wrong answers
         shwrght=n10     $$(opt.)show correct answer
         whichu=n11      $$stores number of last unit entered
         times=n12       $$counts times through unit
         segment,authans=n13,6   $$storage for author answers
         segment,stuans=n14,6    $$storage for student answers
calc     arowloc⇐602
         matches⇐8
         choices⇐12
         ansall⇐1
         shwper⇐1
         shwrong⇐1
         shwrght⇐1
pack     n13,abcdefgh
write    <at,107>This is a matching test on (state relation-
         ship between two columns).  Some items may be used
         more than once.  When you have finished, press -DATA-
         and the test will be evaluated.
at       710

            ( )-1)                    A)
            ( )-2)                    B)
            ( )-3)                    C)
            ( )-4)                    D)
            ( )-5)                    E)
            ( )-6)                    F)
            ( )-7)                    G)
            ( )-8)                    H)
                                      I)
                                      J)
                                      K)
                                      L)
jump     testing
*
*
unit     testing
inhibit  erase
join     cleanup
join     set
join     quesno=matches+1,reset,x
join     test
jump     testing
```


DEFINE
TESTING (DRIVER)
CLEANUP
SET
TEST
RESET
DATA
CHECK
GRADING (DRIVER)
ALLFILL
ASSIGN
SCORE
WRONG
PERCENT

The following routine is divided into two parts. In the first part, the student matches two columns of concepts. In the second part, the student's answers are evaluated and various information is fed back depending on author specifications.

In the context of this documentation, all variable names will be italicized, all unit names (not already preceded by the word "unit") will be in quotation marks, and all commands will be in hyphens. Since each command is documented only with regard to its specific use in this routine, specific functions might not hold true in other contexts.

unit define

1. -inhibit- prevents the current display from being erased when a main unit is encountered.

   Normally, the system erases the screen each time the main unit is changed (by a key press, -jump-, etc.). If the author wants a display to remain on the screen, he/she must place an -inhibit- erase in the unit in which the display is first executed, usually at the beginning of the unit. The -inhibit- effect is not permanent, and will only work the first time a new unit is encountered. After that, the system will resume normal erasing.

2. -zero- sets variables n1 through n14 equal to zero.

   This insures that all variables used in the routine are equal to zero when the first unit is entered.

3. -define- gives meaningful names to variables n1 through n12.

   When variables are identified by number (n1, n2, n35, etc.) it can be difficult to remember what information is stored in which variables. Therefore, an author can give each variable a more meaningful name, just as a parent gives a child a specific name (rather than child1, child2, child3, etc.). These new names may be used in any tag, expression, calculation, etc. in place of the numerical name. For example, in this context

```
calc    quesno⇐Ø
```

is exactly the same as

```
calc    n2⇐Ø
```

   the only difference being that the name *quesno* has more meaning for the author than the name n1.

   Variables act as storage containers for information. If the author wishes to keep track of the number of times a student performs a certain task, the number of correct responses, the position of an arrow--any type

of information--the author designates certain variables as storage bins for the information. The author, however, has a number of choices as to how the information should be stored.

Imagine that you have 10 books and 10 boxes in which to store them. Each individual box is large enough to hold all 10 books. You might choose to put 1 book in each box and pack the leftover space in each box with bits of crumpled paper. However, to make maximum use of space, you might decide to put all 10 books in 1 box. An author may also choose the way in which the system will store information.

All data on the PLATO system is reduced to a series of zeros and ones, (binary notation) regardless of whether the data is a single number or character, or a string of numbers or characters. For example, the following data would be converted and stored thusly;

1 = ØØØØØ1
12 = ØØ11ØØ
t = Ø1Ø1ØØ
the = Ø1Ø1ØØØØ1ØØØØØØ1Ø1

The smallest unit of computer storage is called a bit, just as the smallest whole unit of length is called an inch. Each bit contains a Ø or 1. Just as inches can be grouped and expressed in different, yet equivalent units, so bits can be grouped and expressed in different units. Just as inches can be converted to feet or yards, bits can be converted to characters or words.

36 inches = 3 feet = 1 yard
60 bits = 10 characters = 1 word

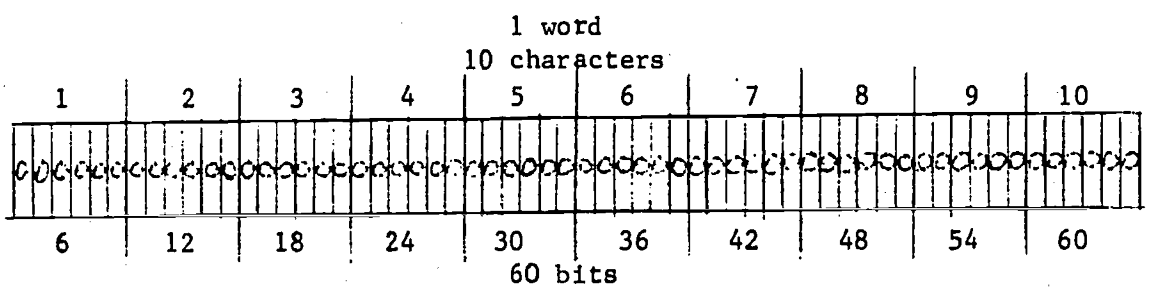


figure 1

On the TUTOR system there are 9,000 bits available to an author for storage purposes. Since an author would have difficulty keeping track of 9,000 bits, these bits are grouped such that every group (array) of 60 bits is a neat package called a variable and is given a numerical name (n1, v14, n37, v142, etc.) Thus, variable names are actually names for a specific group (array) of bits which will always number 60.

Recall the analogy of the books stored in boxes. A similar situation exists when storing information on PLATO. An author can store 1 piece of information in each variable. However, if each piece of information is small (perhaps 1 character -6 bits- in length), a large number

of bits are left over. For example, in this routine we are storing student responses consisting of 1 letter each, requiring 1 character (6 bits) of storage. If each response were stored in a separate variable, each variable would have 54 unused bits filled with zeros.

A more efficient means of storage would be to store each response in a successive group of bits (rather than a successive group of variables). Figures 2 and 3 illustrate this effect. (The actual character is used here rather than the binary equivalent. Remember, however, that each character is actually a series of zeros and ones.)

n1	b	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n2	d	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n3	x	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n4	m	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n5	s	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n6	t	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

figure 2

n1	b	d	x	m	s	t	Ø	Ø	Ø	Ø
n2	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n3	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n4	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n5	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n6	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

figure 3

In figure 1, 6 variables are partially used, while in figure 2 only 6 characters are used. Recall that each variable is an array (group) of bits, arbitrarily set at 60 (10 characters, 6 bits per character) and named n1, n2, v14, etc. The -segment- feature enables an author to redefine his/her own array of bits and store information into successive bit groups (rather than successive whole variables).

```
define     segment,authans=n13,6
              array name   starting    length of each
                           location      bit group
```

The preceding -define- tells the system that, rather than storing data in groups of 60 bits (whole variables), bit segments of 6 bits each will be used, that the starting location will be the left most bit in n13, and that when all the necessary data is collected the total array of bits will be called *authans*. Thus, the system stores data in every 6 bit group starting at n13. When *authans* is referred to throughout the lesson, it will be necessary to specify which segment

of bits in *authans* is being referred to since each segment will contain a different piece of information.

authans

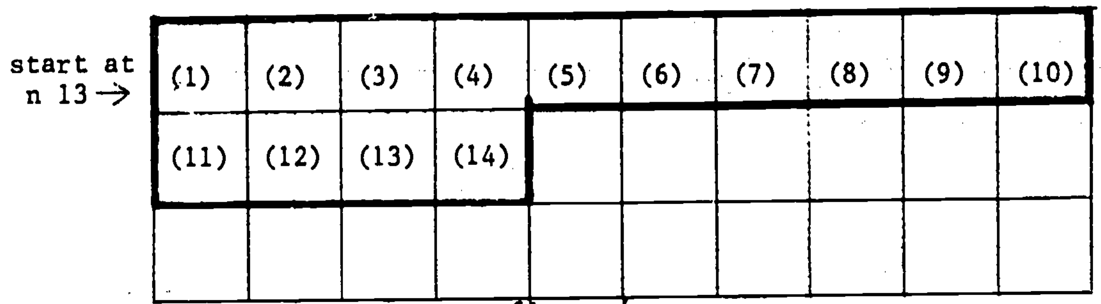


figure 4

In figure 4, 14 pieces of information were collected. These 14 groups of bits are called *authans* and can be referenced as *authans*(1), *authans*(5), *authans*(13), etc. In our routine we refer to *authans(quesno)*. Since we are keeping track of the question number in *quesno*, the system would simply look at the value in *quesno* and refer to that group of bits in *authans*. For example, if the current question number is 4, the value in *quesno* is 4 and *authans (quesno)* refers to the fourth 6 bit segment of *authans*.

Note that n13 is only meaningful as a point at which the system should start segmenting groups of bits. The name *authans* is <u>not</u> intended to replace the name n13, and the 2 names are <u>not</u> interchangeable (as they would be under a regular -define-). Just as n52 is the name for a specific array of bits 60 bits long, *authans* is the name for a specific array of bits of indeterminant length.

The second -segment- is identical to the first. It is the place at which student responses are stored in 6 bit segments, beginning at the left most bit in n14, and named *stuans* for future reference.

4. -calc- gives specific numerical values to the variables in the tag.

These values are set by each author according to the specific needs of his/her lesson. Note that the last 4 variables are set to 1 or Ø. These are options the author may turn on (1) or off (Ø).

5. -pack- stores the character string "abcdefgh" in n13, beginning at the left most bit.

The character string represents the author's answers for the matching routine. Each author should substitute his/her own answers. Note that the variable in the first part of the tag (n13) is really the starting location for *authans*. Thus, *authans* now cantains the character string "abcdefgh". Because of the peculiarities of the -pack- command, the segmented variable name *authans* cannot be used in the -pack- tag. Everywhere else, however, *authans* will be used.

6. -write- tells the system to display material on the student's screen.

This displays the two matching lists on the student's screen. Because of the -inhibit- erase, this display will remain on the screen the first time a new main unit is encountered.

Appendix V

Summary of TUTOR Training Materials Available -- January 1976

1. The MTC basic TUTOR training package consists of interleaved hard-copy explanations and exercises (15) coordinated with the on-line materials "teach", the "tutor" series, the "pptest" series, "ppcode", "ppsoln", "display", "variables", and the "computer" series. A total of 43 commands are described in "tutor", 40 are explained in hardcopy, and 28 are used in the exercises. The materials were designed to be only very basic and to be understood by an audience with weak mathematics backgrounds. Authors typically spend from 5-7 working days from the time they arrive at CERL until they complete the 15 exercises. As noted above, topics other than TUTOR are also being taught during this period.

In addition, the following materials were created and are used for advanced TUTOR training: "execute", "datacollct", the random arrow drill driver, and the matching test driver.

2. The PSO Introduction to TUTOR is a two-part manual. The first part of the manual interleaves comparatively long (relative to the MTC package) sections of reading with six short sets of exercises. The exercises require the author to copy into his lesson space pre-specified passages of TUTOR coding, then try out the product. The PSO manual thus takes an example-presenting approach rather than MTC's problem-solving approach. The first seven sections, roughly the first part of the manual, are accompanied by on-line reading in lesson "introtutor", which further describes commands and provides working examples of them. The manual also offers student mode versions of the exercises from the manual. The latter is not exactly analogous to "ppsoln", in that there are no problems to be solved by the new author using the PSO manual.

This package uses the "conventional" order of teaching commands; answer judging is introduced immediately.

The second part of the manual contains readings without on-line examples or student mode versions of the exercises. The manual contains more than a dozen appendices as well as an alphabetical index. No tests are provided in the manual or in "introtutor". The use of the TUTOR editor is discussed in an appendix of the manual; there are no exercises or practice available. There are no command simulators analogous to "tutor", but "introtutor" provides many working examples. Compared to the MTC materials, the PSO package covers more commands (60-70) in the same or greater depth with fewer exercises and less testing. The average completion time for the manual is about ten working days for authors not currently attending or teaching higher education. Authors at the ARPA/PLATO sites where the authoring staff was chosen from the regular technical training instructor staff could not understand the PSO materials sufficiently well. At other ARPA/PLATO sites (which used either graduate students or experienced CBE staff as authors), the PSO package could have been used.

3. Sherwood's The TUTOR Language is by far the most complete and thorough manual which attempts to teach TUTOR. It is extremely well-knit and manages to convey a vast amount of information quite concisely. However, it is designed to be used by someone who is already familiar with the basics of editing and authoring. It is not a manual for a user with less than six month's experience who lacks a computer or mathematical background. The MTC group has frequently used the Sherwood manual for advanced self-directed training, but has found only a few authors for

whom this manual was suitable as a first course. The book contains no on-line materials. It is indexed and contains a few short appendices. This manual's strengths include careful comparisons and contrasts between similar commands and clever, but useful examples. Furthermore, in many cases, details of the system architecture are related to the form and suggested use of commands.

4. Silas Warner's "introedit" program is the only author mode simulation with which the MTC group has obtained success. Unfortunately, it became available only near the end of the training schedule of ARPA/PLATO sites and was not carefully evaluated. This lesson teaches only basic editing, as does "teach", but is quite thorough. Its apparent success in an area where others have failed is probably the result of the constant implicit and explicit cues that remind the new author that this lesson is a simulation of the real TUTOR editor.

Appendix VI

The Use of Indexed Variables by Authors

The extent of use of powerful programming techniques is difficult to determine precisely without spending a great deal of time carefully examining many programs. Furthermore, if the programming is not carefully documented, the analysis may consume more time than was spent to create the code. A comprehension test can demonstrate that an author understands a particular programming technique or possibly even that he can use it; it does not indicate that he actually uses the technique, however. Finally, true mastery of a concept is more reliably and usefully demonstrated by the actual implementation of the concept.

In order to estimate the level of usage of the calculational portions of the PLATO system by ARPA authors, we chose to count the occurrences of indexing, a powerful, general programming technique. A non-indexed variable contains a number useful to the program; an indexed variable contains a number which specifies the location of a number useful to the program. Use of a variable as an index (as a "pointer") is necessary for a whole host of operations which may be found within instructional lessons employing complex strategies (drill paradigms, simulations, etc.) and for a broad range of tasks unrelated to lesson strategies (recording successive answers, displaying and judging computer-generated questions, etc.). Indexed variables are also virtually mandated by many non-instructional uses: information storage and retrieval, student routing, data management programs.

Any use of indexed variables requires the author to conform to formats which allow a computer-search of the lesson to quickly identify

the type of use and frequency of occurrence. Several other calculational features possess this "computer-searchable" aspect, but few seem so generally necessary for a variety of useful tasks. For example, use of matrices and arrays implies sophistication with calculational features, but absence of such use does not imply computational naïveté. Other features are so new that "old" lessons could not be compared on an equal basis. Finally, features such as "looping" can be implemented in so many different ways that analysis-by-search is impractical. Indexed variables (or the concept of pointers) can be implemented in two ways: explicitly (and unambiguously) or coincidentally with the "segment" feature. Since segments are also used to save space or to permit bit manipulation, their presence may or may not imply the same things that indexed variables do. For that reason, both forms of implementation are enumerated separately below. A third, highly complex method of implementing the concept of indexing with other commands and syntaxes was not found.

It must be realized that the presence or absence of indexed variables in a lesson is probably quite unrelated to its educational effectiveness. All that can be concluded is that a lesson without indexed variables cannot perform a set of operations which MIGHT be useful instructionally.

With these caveats, the following data is presented. It was gathered by searching instructional lessons for the strings "nc(", "vc(", "n(", "v(", and "segment,". Named segments found were then used as search targets. For example, if a segment named "time" was found, a search for the string "time(" was made. The computer search retrieved the whole TUTOR statement containing the potential example

of indexing. If true indexing was indicated [e.g., "time(n10)" rather than "time(7)"] the line was counted. No line was counted more than once no matter how many examples of indexing it contained. No data management or routing lessons were examined. In both Chanute and Sheppard lessons the indexed variables used in a one-per-lesson, standardized data collection routine were not counted. Thus, within the limits of the searching techniques, the counts of use of indexed variables below reflects their use for instruction.

Explicitly Defined Indexing--e.g., n(n10)

Site	Total lessons checked	Lessons w/ explicitly indx vars	Total # commands found	% Lessons with indx vars	Commands per lesson
Chanute	39	5	6	13%	1.2
Aberdeen	24	8	46	33%	5.8
Sheppard	71	19	90	28%	4.7

Indexing by Means of Segments--e.g., time(n10)

Site	Total lessons checked	Lessons w/ segmented indx vars	Total # commands found	% Lessons with indx vars	Commands per lesson
Chanute	39	37	many	95%	many
Chanute[a]	39	10	44	26%	1.1
Aberdeen	24	3	45	12%	2.1
Sheppard	50	16	130	32%	8.1

[a]This entry ignores indexed variables which were present only in data collection and matching test routines. In general, these routines were not created by the author of the lesson, but rather were the work of two or three authors with specialized roles. Hence it seems a more valid assessment of how widespread among authors was the use of indexed variables.

Appendix VII

Training Standards for Basic TUTOR

This appendix contains a partial list of objectives for six topics which a new author can be expected to attain during basic TUTOR training. Also included is an estimate of the inter-quartile range for the authors described in Table 1 in the body of this report.

Editing

Prerequisites--A PLATO demonstration and one-half hour or more student mode experience.

Using only PLATO-supplied help, an author, given the name of an empty lesson, will 1) sign on; 2) fill out the lesson data page; 3) create a new block; 4) type in, with correct tabulation, a supplied piece of coding; 5) execute that code as a student; 6) return to author mode to delete and replace pre-specified lines. False attempts are permitted but the task must be completed within 5 minutes.

Time estimate: 3-9 hours.

Display

Prerequisite--Editing

Given a display that can be created with 30 -circle- (including arcs), -size-, -rotate-, -write-, -at-, and -draw- commands, the author will produce a reasonable facsimile of the display within 30 minutes. The copy shall be faithful to the original with a tolerance of 16 dots (.25") for each point.

Given a location on the plasma panel, the author will give its position in course grid (±25 dots) 75% of the time.

Time estimate: 2-3 hours.

Calculation

Prerequisite--Display

Given a series of sequentially (i.e., linearly) executed -calc- commands (including *, ÷, +, -, multiple assignments, on-level deep indexed variables), the author will correctly predict the value of 90%

of the calculations.

Given a series of desired outcomes paired with arithmetic* conditions, the author will construct a conditional command to select the outcomes with 100% accuracy.

Given a series of numerical values to be displayed in various formats, the author will correctly describe the output for 80% of the cases.

Time estimate: 1-12 hours.

*i.e., Not logical (>, =, and, etc.) conditions, but arithmetic (if condition = 1, do this; if 2, do that;. . .).

<u>Looping</u>

Prerequisite--Calculation and Branching

Given displays which can be iteratively generated by means of one -do- loop controlling one display command, the author will provide the coding necessary to produce the displays. The author will be allowed three attempts to solve each problem.

Time estimate: 1-6 hours.

<u>Branching</u>

Prerequisite--Display

Given: -next-, -help-, -goto-, -jump-, -do-, and -term-, the author will identify those commands and keys which 1) begin a new main unit, 2) erase the screen, 3) create a base unit, 4) interrupt linear execution order 5) provide some sort of automatic return from the branch.

Given a list of branching commands that provide some sort of automatic return, the author will specify the conditions and nature of the

return.

Given a program listing for a lesson containing 10 or fewer units and only the student- and author-controlled branching commands -next-, -back-, -help-, -end-, -do- (non-looping), -jump-, and -goto-, plus a -write- command to display the name of each unit, the author will predict with 80% accuracy the displays produced by pressing various sequences of function keys.

Time estimate: 3-9 hours.

Judging

Prerequisites--Display and experience with sequential order of command execution.

Given samples of acceptable and unacceptable student responses (with appropriate feedback for each), the author will generate coding to judge each response correctly and provide the corresponding feedback. Some of the responses shall require -specs- options to work efficiently. The resultant coding will provide accurate feedback and response markup in 90% of the cases.

Time estimate: 5-16 hours.